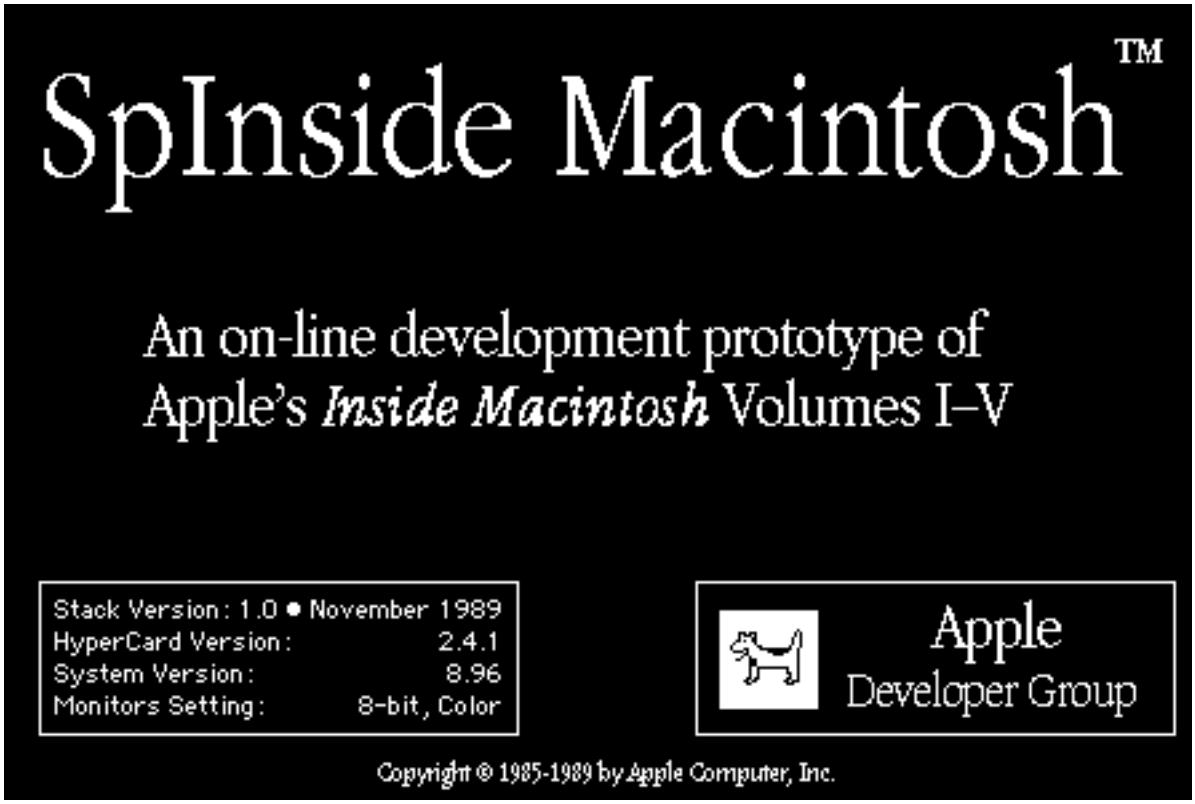

THE APPLETALK MANAGER

About This Chapter
AppleTalk Protocols
AppleTalk Transaction Protocol
    Transactions
    Datagram Loss Recovery
About the AppleTalk Manager
Calling the AppleTalk Manager from Pascal
    Opening and Closing AppleTalk
    AppleTalk Link Access Protocol
        Data Structures
        Using ALAP
        ALAP Routines
        Example
    Datagram Delivery Protocol
        Data Structures
        Using DDP
        DDP Routines
        Example
    AppleTalk Transaction Protocol
        Data Structures
        Using ATP
        ATP Routines
        Example

Name-Binding Protocol
Data Structures
Using NBP
NBP Routines
Example
Miscellaneous Routines
New AppleTalk Manager Pascal Interface
Using Pascal
MPP Parameter Block
ATP Parameter Block
Building Data Structures
Picking a Node Address in the Server Range
Sending Packets to One's Own Node
ATP Driver Changes
Sending an ATP Request Through a Specified Socket
Aborting ATP SendRequests
Aborting ATP GetRequests
Name Binding Protocol Changes
Multiple Concurrent NBP Requests
KillNBP function
Variable Resources
Calling the AppleTalk Manager from Assembly Language
Opening AppleTalk
Example
AppleTalk Link Access Protocol
Data Structures
Using ALAP
ALAP Routines
Datagram Delivery Protocol
Data Structures
Using DDP
DDP Routines
AppleTalk Transaction Protocol
Data Structures
Using ATP
ATP Routines
Name-Binding Protocol
Data Structures
Using NBP
NBP Routines
Extended Protocol Package Driver
Version
Error Reporting
.XPP Driver Functions Overview
Using AppleTalk Name Binding Protocol
Opening and Closing Sessions
Session Maintenance
Commands on an Open Session
Getting Server Status Information
Attention Mechanism
The Attention Routine
Calling the .XPP Driver
Using XPP
Allocating Memory
Opening the .XPP Driver

Example
Open Errors
Closing the .XPP Driver
Close Errors
Session Control Block
How to Access the .XPP Driver
General
.XPP Driver Parameter Block Record
AppleTalk Session Protocol Functions
Note on Result Codes
AFP Implementation
Mapping AFP Commands
AFPCall Function
General Command Format
Login Command Format
AFPWrite Command Format
AFPRead Command Format
CCB Sizes
.XPP Driver Result Codes
Protocol Handlers and Socket Listeners
Data Reception in the AppleTalk Manager
Writing Protocol Handlers
Timing Considerations
Writing Socket Listeners
Summary of the AppleTalk Manager

ABOUT THIS CHAPTER

The AppleTalk Manager is an interface to a pair of RAM device drivers that allow Macintosh programs to send and receive information via an AppleTalk network. This chapter describes the AppleTalk Manager in detail.

The AppleTalk Manager has been enhanced through the implementation of new protocols and an increase in the functionality of the existing interface.

Reader's guide: The AppleTalk Manager provides services that allow Macintosh programs to interact with clients in devices connected to an AppleTalk network. Hence you need the information in this chapter only if your application uses AppleTalk.

The following is a brief summary of the changes that have been made to the AppleTalk Manager interface.

- New parameter block–style Pascal calls have been added for the entire AppleTalk Manager. These new calls give the application programmer better control of AppleTalk operation within an application.
- At open time, the .MPP driver can be told to pick a node number in the server range. This is a more time consuming but more thorough operation than is selecting a node number in the workstation range, and it is required for devices acting as servers.
- Multiple concurrent NBP requests are now supported (just as multiple concurrent ATP requests have been supported). The KillNBP command has been implemented to abort an outstanding NBP request.
- ATP requests can now be sent through client-specified sockets, instead of having ATP pick the socket itself.
- The ability to send packets to one's own node is supported (although this functionality is, in the default case, disabled).
- Two new ATP abort calls have been added: KillSendReq and KillGetReq. KillSendReq is functionally equivalent to RelTCB, although its arguments are different. KillGetReq is a new call for aborting outstanding GetRequests.
- Additional machine-dependent resources have been added to support, for example, more dynamic sockets and more concurrent ATP requests.
- A new protocol called the Echo Protocol (EP) is supported.
- A new driver, .XPP, has been added. The .XPP driver implements the workstation side of the AppleTalk Session Protocol (ASP) and a small portion of the AppleTalk Filing Protocol.

To determine if your application is running on a machine that supports these enhanced features, check the version number of the .MPP driver (at offset DCtlQueue+1 in the Device Control Entry). A version number of 48 (NCVersion) or greater indicates the presence of the new drivers.

You should already be familiar with:

- events, as discussed in the Toolbox Event Manager chapter
- interrupts and the use of devices and device drivers, as described in

the Device Manager chapter, if you want to write your own assembly-language additions to the AppleTalk Manager
- the Inside AppleTalk manual, if you want to understand AppleTalk protocols in detail

APPLETALK PROTOCOLS

The AppleTalk Manager provides a variety of services that allow Macintosh programs to interact with programs in devices connected to an AppleTalk network. This interaction, achieved through the exchange of variable-length blocks of data (known as packets) over AppleTalk, follows well-defined sets of rules known as protocols.

Although most programmers using AppleTalk needn't understand the details of these protocols, they should understand the information in this section—what the services provided by the different protocols are, and how the protocols are interrelated. Detailed information about AppleTalk protocols is available in Inside AppleTalk.

The AppleTalk system architecture consists of a number of protocols arranged in layers. Each protocol in a specific layer provides services to higher-level layers (known as the protocol's clients) by building on the services provided by lower-level layers. A Macintosh program can use services provided by any of the layers in order to construct more sophisticated or more specialized services. Figure 1 shows the AppleTalk Protocols and their corresponding network layers.

The AppleTalk Manager contains the following protocols:

- AppleTalk Link Access Protocol
- Datagram Delivery Protocol
- Routing Table Maintenance Protocol
- Name-Binding Protocol
- AppleTalk Transaction Protocol

The following protocols have been added to the AppleTalk Manager:

- Echo Protocol
- AppleTalk Session Protocol (workstation side)
- AppleTalk Filing Protocol (small portion of the workstation side)

In Figure 1, the lines indicate the interaction between the protocols. Notice that like the Routing Table Maintenance Protocol, the Echo Protocol is not directly accessible to Macintosh programs.

The details of these protocols are provided in Inside AppleTalk.

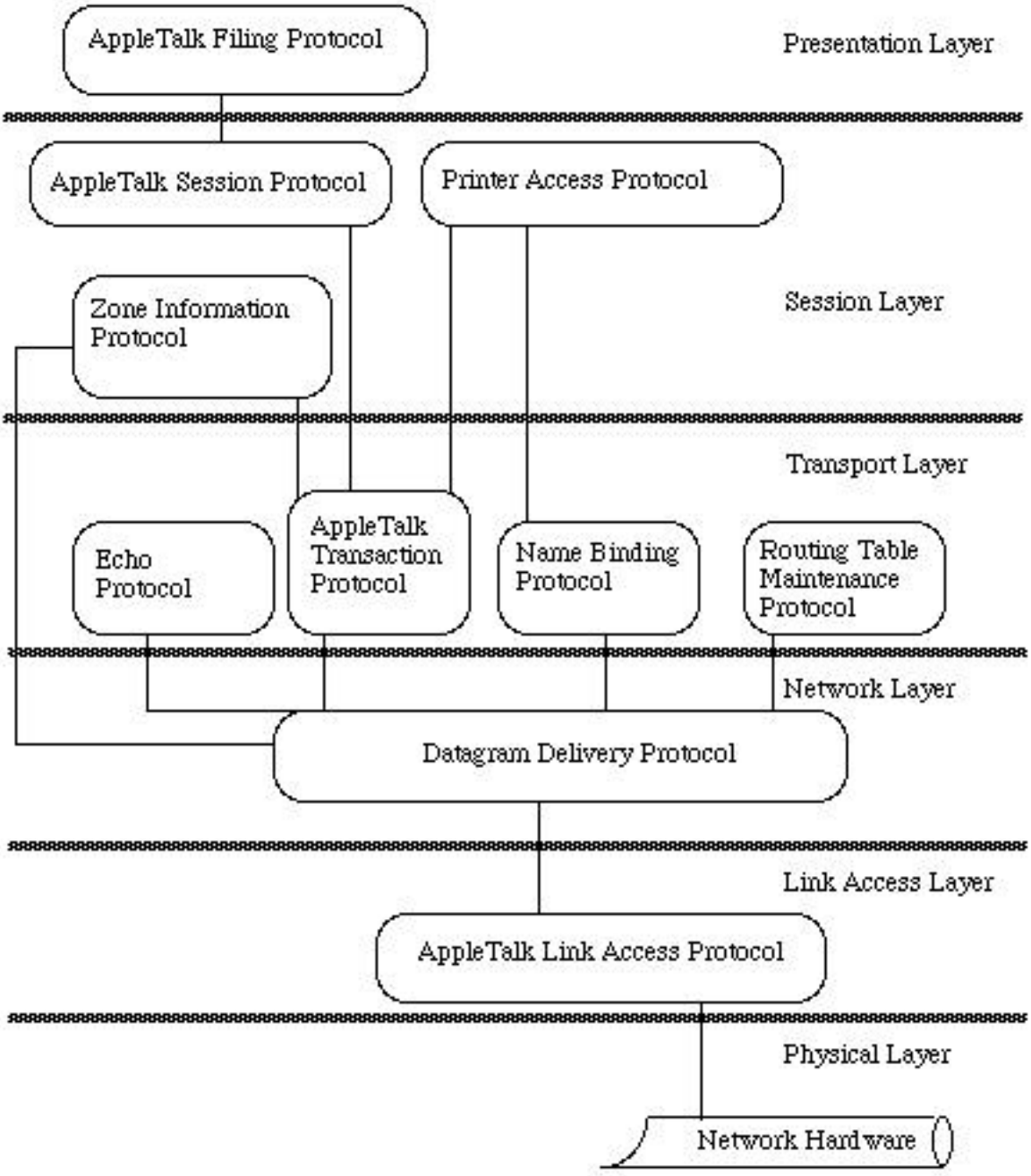

Figure 1–AppleTalk Protocols and OSI Network Layers

Figure 2 illustrates the Macintosh AppleTalk Drivers and the layered structure of the protocols which are accessible through each driver. Note that the Routing Table Maintenance Protocol isn't directly accessible to Macintosh Programs.

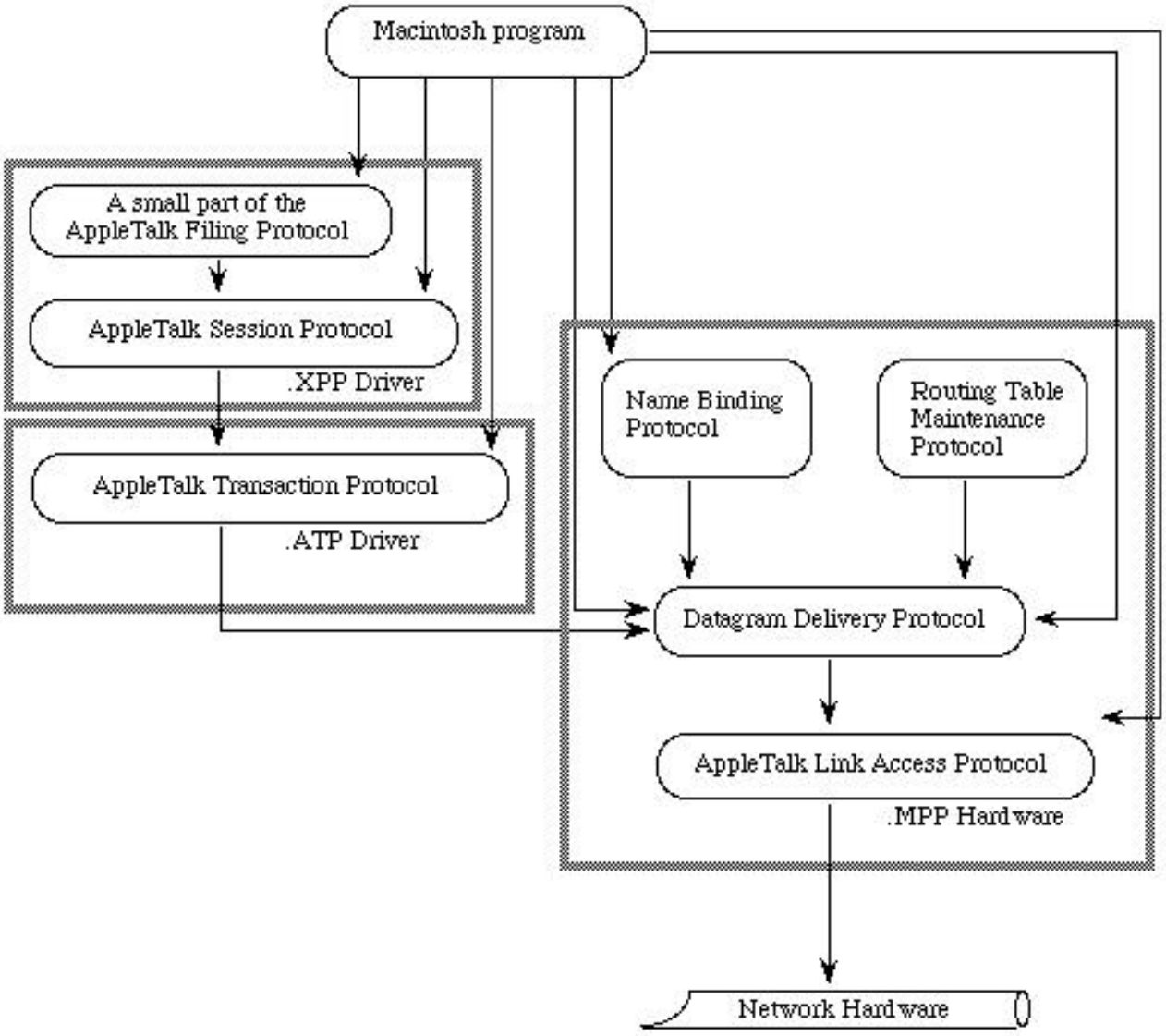

Figure 2–Macintosh AppleTalk Drivers

Figure 2–Macintosh AppleTalk Drivers

The AppleTalk Link Access Protocol (ALAP) provides the lowest-level services of the AppleTalk system. Its main function is to control access to the AppleTalk network among various competing devices. Each device connected to an AppleTalk network, known as a node, is assigned an eight-bit node ID number that identifies the node. ALAP ensures that each node on an AppleTalk network has a unique node ID, assigned dynamically when the node is started up.

ALAP provides its clients with node-to-node delivery of data frames on a single AppleTalk network. An ALAP frame is a variable-length packet of data preceded and followed by control information referred to as the ALAP frame header and frame trailer, respectively. The ALAP frame header includes the node IDs of the

frame's destination and source nodes. The AppleTalk hardware uses the destination node ID to deliver the frame. The frame's source node ID allows a program in the receiving node to determine the identity of the source. A sending node can ask ALAP to send a frame to all nodes on the AppleTalk network; this broadcast service is obtained by specifying a destination node ID of 255.

ALAP can have multiple clients in a single node. When a frame arrives at a node, ALAP determines which client it should be delivered to by reading the frame's ALAP protocol type. The ALAP protocol type is an eight-bit quantity, contained in the frame's header, that identifies the ALAP client to whom the frame will be sent. ALAP calls the client's protocol handler, which is a software process in the node that reads in and then services the frames. The protocol handlers for a node are listed in a protocol handler table.

An ALAP frame trailer contains a 16-bit frame check sequence generated by the AppleTalk hardware. The receiving node uses the frame check sequence to detect transmission errors, and discards frames with errors. In effect, a frame with an error is "lost" in the AppleTalk network, because ALAP doesn't attempt to recover from errors by requesting the sending node to retransmit such frames. Thus ALAP is said to make a "best effort" to deliver frames, without any guarantee of delivery.

An ALAP frame can contain up to 600 bytes of client data. The first two bytes must be an integer equal to the length of the client data (including the length bytes themselves).

Datagram Delivery Protocol (DDP) provides the next-higher level protocol in the AppleTalk architecture, managing socket-to-socket delivery of datagrams over AppleTalk internets. DDP is an ALAP client, and uses the node-to-node delivery service provided by ALAP to send and receive datagrams. Datagrams are packets of data transmitted by DDP. A DDP datagram can contain up to 586 bytes of client data. Sockets are logical entities within the nodes of a network; each socket within a given node has a unique eight-bit socket number.

On a single AppleTalk network, a socket is uniquely identified by its AppleTalk address—its socket number together with its node ID. To identify a socket in the scope of an AppleTalk internet, the socket's AppleTalk address and network number are needed. Internets are formed by interconnecting AppleTalk networks via intelligent nodes called bridges. A network number is a 16-bit number that uniquely identifies a network in an internet. A socket's AppleTalk address together with its network number provide an internet-wide unique socket identifier called an internet address.

Sockets are owned by socket clients, which typically are software processes in the node. Socket clients include code called the socket listener, which receives and services datagrams addressed to that socket. Socket clients must open a socket before datagrams can be sent or received through it. Each node contains a socket table that lists the listener for each open socket.

A datagram is sent from its source socket through a series of AppleTalk networks, being passed on from bridge to bridge, until it reaches its destination network. The ALAP in the destination network then delivers the datagram to the node containing the destination socket. Within that node the datagram is received by ALAP calling the DDP protocol handler, and by the DDP protocol handler in turn calling the destination socket listener, which for most

applications will be a higher-level protocol such as the AppleTalk Transaction Protocol.

Bridges on AppleTalk internets use the Routing Table Maintenance Protocol (RTMP) to maintain routing tables for routing datagrams through the internet. In addition, nonbridge nodes use RTMP to determine the number of the network to which they're connected and the node ID of one bridge on their network. The RTMP code in nonbridge nodes contains only a subset of RTMP (the RTMP stub), and is a DDP client owning socket number 1 (the RTMP socket).

Socket clients are also known as network-visible entities, because they're the primary accessible entities on an internet. Network-visible entities can choose to identify themselves by an entity name, an identifier of the form

```
object:type@zone
```

Each of the three fields of this name is an alphanumeric string of up to 32 characters. The object and type fields are arbitrary identifiers assigned by a socket client, to provide itself with a name and type descriptor (for example, abs:Mailbox). The zone field identifies the zone in which the socket client is located; a zone is an arbitrary subset of AppleTalk networks in an internet. A socket client can identify itself by as many different names as it chooses. These aliases are all treated as independent identifiers for the same socket client.

The Name-Binding Protocol (NBP) maintains a names table in each node that contains the name and internet address of each entity in that node. These name-address pairs are called NBP tuples. The collection of names tables in an internet is known as the names directory.

NBP allows its clients to add or delete their name-address tuples from the node's names table. It also allows its clients to obtain the internet addresses of entities from their names. This latter operation, known as name lookup (in the names directory), requires that NBP install itself as a DDP client and broadcast special name-lookup packets to the nodes in a specified zone. These datagrams are sent by NBP to the names information socket—socket number 2 in every node using NBP.

NBP clients can use special meta-characters in place of one or more of the three fields of the name of an entity it wishes to look up. The character "=" in the object or type field signifies "all possible values". The zone field can be replaced by "*", which signifies "this zone"—the zone in which the NBP client's node is located. For example, an NBP client performing a lookup with the name

```
=:Mailbox@*
```

will obtain in return the entity names and internet addresses of all mailboxes in the client's zone (excluding the client's own names and addresses). The client can specify whether one or all of the matching names should be returned.

NBP clients specify how thorough a name lookup should be by providing NBP with the number of times (retry count) that NBP should broadcast the lookup packets and the time interval (retry interval) between these retries.

As noted above, ALAP and DDP provide "best effort" delivery services with no recovery mechanism when packets are lost or discarded because of errors. Although for many situations such a service suffices, the AppleTalk Transaction Protocol (ATP) provides a reliable loss-free transport service. ATP uses transactions, consisting of a transaction request and a transaction response, to deliver data reliably. Each transaction is assigned a 16-bit transaction ID number to distinguish it from other transactions. A transaction request is retransmitted by ATP until a complete response has been received, thus allowing for recovery from packet-loss situations. The retry interval and retry count are specified by the ATP client sending the request.

Although transaction requests must be contained in a single datagram, transaction responses can consist of as many as eight datagrams. Each datagram in a response is assigned a sequence number from 0 to 7, to indicate its ordering within the response.

ATP is a DDP client, and uses the services provided by DDP to transmit requests and responses. ATP supports both at-least-once and exactly-once transactions. Four of the bytes in an ATP header, called the user bytes, are provided for use by ATP's clients—they're ignored by ATP.

ATP's transaction model and means of recovering from datagram loss are covered in detail below.

The Echo Protocol (EP) provides an echoing service through static socket number 4 known as the echoer socket. The echoer listens for packets received through this socket. Any correctly formed packet sent to the echoer socket on a node will be echoed back to its sender.

This simple protocol can be used for two important purposes:

- EP can be used by any Datagram Delivery Protocol (DDP) client to determine if a particular node (known to have an echoer) is accessible over an internet.
- EP is useful in determining the average time it takes for a packet to travel to a remote node and back. This is very helpful in developing client-dependent heuristics for estimating the timeouts to be specified by clients of ATP, ASP, and other protocols.

Programs cannot access EP directly via the AppleTalk Manager. The EP implementation exists solely to respond to EP requests sent by other nodes. EP is a DDP client residing on statically-assigned socket 4, the echoing socket. Clients wishing to send EP requests (and receive EP responses) should use the Datagram Delivery Protocol (DDP) to send the appropriate packet. For more information about the EP packet format, see Inside AppleTalk.

The AppleTalk Session Protocol (ASP) provides for the setting up, maintaining and closing down of a session. A session is a logical relationship between two network entities, a workstation and a server. The workstation tells the server what to do, and the server responds with the appropriate actions. ASP makes sure that the session dialog is maintained in the correct sequence and that both ends of the conversation are properly participating.

ASP will generally be used between two communicating network entities where one is providing a service to the other (for example, a server is providing a

service to a workstation) and the service provided is state-dependent. That is, the response to a particular request from an entity is dependent upon other previous requests from that entity. For example, a request to read bytes from a file is dependent upon a previous request to open that file in the first place. However, a request to return the time of day is independent of all such previous requests.

When the service provided is state-dependent, requests must be delivered to the server in the same order as generated by the workstation. ASP guarantees requests are delivered to the server in the order in which they are issued, and that duplicate requests are never delivered (another requirement of state-dependent service).

ASP is an asymmetric protocol, providing one set of services to the workstation and a different set of services to the server.

ASP workstation clients initiate (open) sessions, send requests (commands) on that session, and close sessions down. ASP server clients receive and respond (through command replies) to these requests. ASP guarantees that these requests are delivered in the same order as they are made, and without duplication. ASP is also responsible for closing down the session if one end fails or becomes unreachable, and will inform its client (either server or workstation) of the action.

ASP also provides various additional services, such as allowing a workstation to obtain server status information without opening a session to a server, writing blocks of data from the workstation to the server end of the session, and providing the ability for a server to send an attention message to the workstation.

ASP assumes that the workstation client has a mechanism for looking up the network address of the server with which it wants to set up a session. (Generally this is done using the AppleTalk Name Binding Protocol.)

Both ends of the session periodically check to see that the other end of the session is still responsive. If one end fails or becomes unreachable the other end closes the session.

ASP is a client of ATP and calls ATP for transport services.

ASP does not

- ensure that consecutive commands complete in the order in which they were sent (and delivered) to the server
- understand or interpret the syntax or the semantics of the commands sent to the server by the workstation
- allow the server to send commands to the workstation (The server is allowed to alert the workstation through the server's attention mechanism only.)

Note: The .XPP driver does implement the workstation side of the AppleTalk Filing Protocol login command.

The AppleTalk Filing Protocol (AFP) allows a workstation on an AppleTalk network to access files on an AFP file server. AFP specifies a remote filing system

that provides user authentication and an access control mechanism that supports volume and folder-level access rights. For details of AFP, refer to Inside AppleTalk.

APPLETALK TRANSACTION PROTOCOL

This section covers ATP in greater depth, providing more detail about three of its fundamental concepts: transactions, buffer allocation, and recovery of lost datagrams.

Transactions

A transaction is a interaction between two ATP clients, known as the requester and the responder. The requester calls the .ATP driver in its node to send a transaction request (TReq) to the responder, and then awaits a response. The TReq is received by the .ATP driver in the responder's node and is delivered to the responder. The responder then calls its .ATP driver to send back a transaction response (TResp), which is received by the requester's .ATP driver and delivered to the requester. Figure 3 illustrates this process.

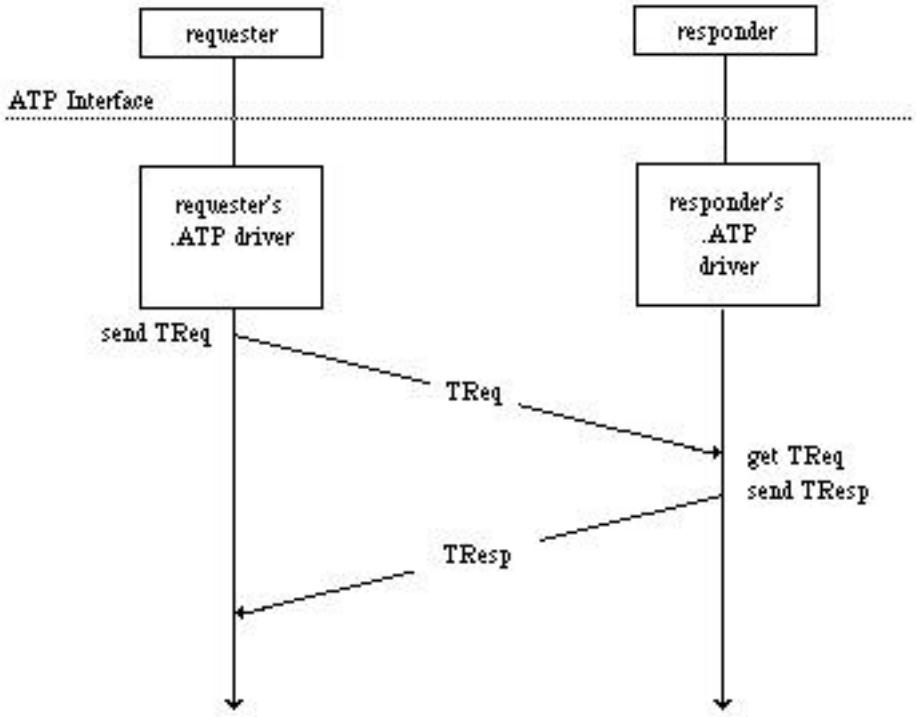

Figure 3–Transaction Process

Figure 3–Transaction Process

Simple examples of transactions are:

- read a counter, reset it and send back the value read
- read six sectors of a disk and send back the data read
- write the data sent in the TReq to a printer

A basic assumption of the transaction model is that the amount of ATP data sent in the TReq specifying the operation to be performed is small enough to fit in a single datagram. A TResp, on the other hand, may span several datagrams, as in the second example. Thus, a TReq is a single datagram, while a TResp consists of up to eight datagrams, each of which is assigned a sequence number from 0 to 7 to indicate its position in the response.

The requester must, before calling for a TReq to be sent, set aside enough buffer space to receive the datagram(s) of the TResp. The number of buffers allocated (in other words, the maximum number of datagrams that the responder can send) is indicated in the TReq by an eight-bit bit map. The bits of this bit map are numbered 0 to 7 (the least significant bit being number 0); each bit corresponds to the response datagram with the respective sequence number.

Datagram Loss Recovery

The way that ATP recovers from datagram loss situations is best explained by an example; see Figure 4. Assume that the requester wants to read six sectors of 512 bytes each from the responder's disk. The requester puts aside six 512-byte buffers (which may or may not be contiguous) for the response datagrams, and calls ATP to send a TReq. In this TReq the bit map is set to binary 00111111 or decimal 63. The TReq carries a 16-bit transaction ID, generated by the requester's .ATP driver before sending it. (This example assumes that the requester and responder have already agreed that each buffer can hold 512 bytes.) The TReq is delivered to the responder, which reads the six disk sectors and sends them back, through ATP, in TResp datagrams bearing sequence numbers 0 through 5. Each TResp datagram also carries exactly the same transaction ID as the TReq to which they're responding.

There are several ways that datagrams may be lost in this case. The original TReq could be lost for one of many reasons. The responding node might be too busy to receive the TReq or might be out of buffers for receiving it, there could be an undetected collision on the network, a bit error in the transmission line, and so on. To recover from such errors, the requester's .ATP driver maintains an ATP retry timer for each transaction sent. If this timer expires and the complete TResp has not been received, the TReq is retransmitted and the retry timer is restarted.

A second error situation occurs when one or more of the TResp datagrams isn't received correctly by the requester's .ATP driver (datagram 1 in Figure 4). Again, the retry timer will expire and the complete TResp will not have been received; this will result in a retransmission of the TReq. However, to avoid unnecessary retransmission of the TResp datagrams already properly received, the bit map of this retransmitted TReq is modified to reflect only those datagrams not yet received. Upon receiving this TReq, the responder retransmits only the missing response datagrams.

Another possible failure is that the responder's .ATP driver goes down or the responder becomes unreachable through the underlying network system. In this

case, retransmission of the TReq could continue indefinitely. To avoid this situation, the requester provides a maximum retry count; if this count is exceeded, the requester's .ATP driver returns an appropriate error message to the requester.

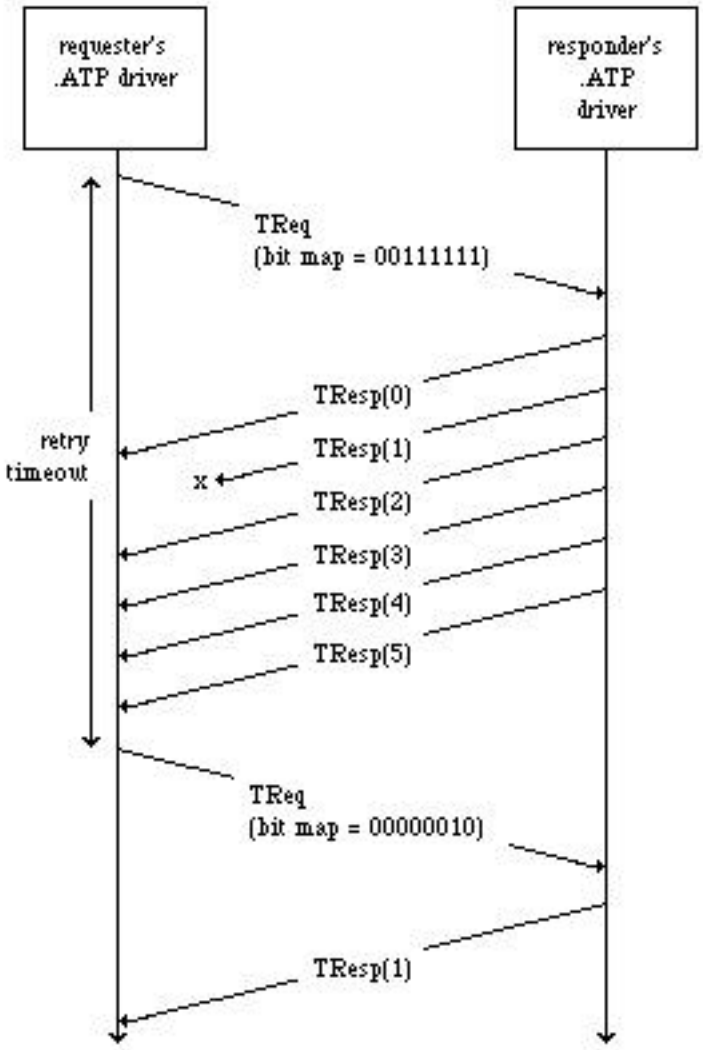

Figure 4–Datagram Loss Recovery

Figure 4–Datagram Loss Recovery

Note: There may be situations where, due to an anticipated delay, you'll want a request to be retransmitted more than 255 times; specifying a retry count of 255 indicates "infinite retries" to ATP and will cause a message to be retransmitted until the request has either been serviced, or been cancelled through a specific call.Finally, in our example, what if the responder is able to provide only four disk sectors (having reached the end of the disk) instead of the six requested? To handle this

situation, there's an end-of-message (EOM) flag in each TResp datagram. In this case, the TResp datagram numbered 3 would come with this flag set. The reception of this datagram informs the requester's .ATP driver that TResps numbered 4 and 5 will not be sent and should not be expected.

When the transaction completes successfully (all expected TResp datagrams are received or TResp datagrams numbered 0 to n are received with datagram n's EOM flag set), the requester is informed and can then use the data received in the TResp.

ATP provides two classes of service: at-least-once (ALO) and exactly-once (XO). The TReq datagram contains an XO flag that's set if XO service is required and cleared if ALO service is adequate. The main difference between the two is in the sequence of events that occurs when the TReq is received by the responder's .ATP driver.

In the case of ALO service, each time a TReq is received (with the XO flag cleared), it's delivered to the responder by its .ATP driver; this is true even for retransmitted TReqs of the same transaction. Each time the TReq is delivered, the responder performs the requested operation and sends the necessary TResp datagrams. Thus, the requested operation is performed at least once, and perhaps several times, until the transaction is completed at the requester's end.

The at-least-once service is satisfactory in a variety of situations—for instance, if the requester wishes to read a clock or a counter being maintained at the responder's end. However, in other circumstances, repeated execution of the requested operation is unacceptable. This is the case, for instance, if the requester is sending data to be printed at the responding end; exactly-once service is designed for such situations.

The responder's .ATP driver maintains a transactions list of recently received XO TReqs. Whenever a TReq is received with its XO flag set, the driver goes through this list to see if this is a retransmitted TReq. If it's the first TReq of a transaction, it's entered into the list and delivered to the responder. The responder executes the requested operation and calls its driver to send a TResp. Before sending it out, the .ATP driver saves the TResp in the list.

When a retransmitted TReq for the same XO transaction is received, the responder's .ATP driver will find a corresponding entry in the list. The retransmitted TReq is not delivered to the responder; instead, the driver automatically retransmits the response datagrams that were saved in the list. In this way, the responder never sees the retransmitted TReqs and the requested operation is performed only once.

ATP must include a mechanism for eventually removing XO entries from the responding end's transaction list; two provisions are made for this. When the requester's .ATP driver has received all the TResp datagrams of a particular transaction, it sends a datagram known as a transaction release (TRel); this tells the responder's .ATP driver to remove the transaction from the list. However, the TRel could be lost in the network (or the responding end may die, and so on), leaving the entry in the list forever. To account for this situation, the responder's .ATP driver maintains a release timer for each transaction. If this timer expires and no activity has occurred for the transaction, its entry is removed from the transactions list.

The AppleTalk Manager is divided into three parts (see Figure 5):

- A lower-level driver called ".MPP" that contains code to implement ALAP, DDP, NBP, and the RTMP stub; this includes separate code resources loaded in when an NBP name is registered or looked up.
- A higher-level driver called ".ATP" that implements ATP.
- A Pascal interface to these two drivers, which is a set of Pascal data types and routines to aid Pascal programmers in calling the AppleTalk Manager.

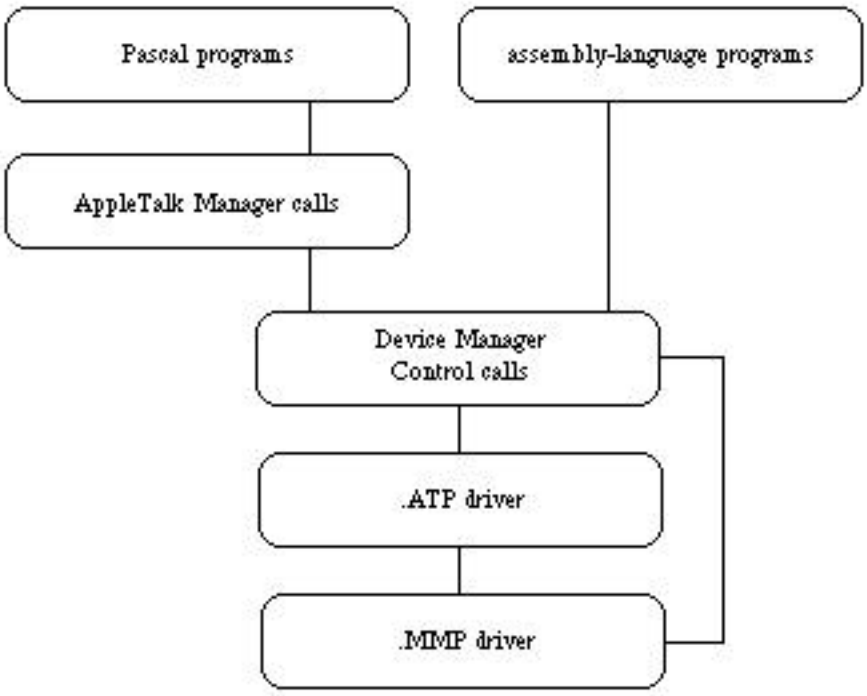

Figure 5–Calling the AppleTalk Manager

Figure 5–Calling the AppleTalk Manager

The two drivers and the interface to them are not in ROM; your application must link to the appropriate object files.

Pascal programmers make calls to the AppleTalk Manager's Pascal interface, which in turn makes Device Manager Control calls to the two drivers. Assembly-language programmers make Device Manager Control calls directly to the drivers.

Note: Pascal programmers can, of course, make PBControl calls directly if they wish.

The AppleTalk Manager provides ALAP routines that allow a program to:

- send a frame to another node
- receive a frame from another node
- add a protocol handler to the protocol handler table
- remove a protocol handler from the protocol handler table

Each node may have up to four protocol handlers in its protocol handler table, two of which are currently used by DDP.

By calling DDP, socket clients can:

- send a datagram via a socket
- receive a datagram via a socket
- open a socket and add a socket listener to the socket table
- close a socket and remove a socket listener from the socket table

Each node may have up to 12 open sockets in its socket table.

Programs cannot access RTMP directly via the AppleTalk Manager; RTMP exists solely for the purpose of providing DDP with routing information.

The NBP code allows a socket client to:

- register the name and socket number of an entity in the node's names table
- determine the address (and confirm the existence) of an entity
- delete the name of an entity from the node's names table

The AppleTalk Manager's .ATP driver allows a socket client to do the following:

- open a responding socket to receive requests
- send a request to another socket and get back a response
- receive a request via a responding socket
- send a response via a responding socket
- close a responding socket

Note: Although the AppleTalk Manager provides four different protocols for your use, you're not bound to use all of them. In fact, most programmers will use only the NBP and ATP protocols.

AppleTalk communicates via channel B of the Serial Communications Controller (SCC). When the Macintosh is started up with a disk containing the AppleTalk code, the status of serial port B is checked. If port B isn't being used by another device driver, and is available for use by AppleTalk, the .MPP driver is loaded into the system heap. On a Macintosh 128K, only the MPP code is loaded at system startup; the .ATP driver and NBP code are read into the application heap when the appropriate commands are issued. On a Macintosh 512K or XL, all AppleTalk code is loaded into the system heap at system startup.

After loading the AppleTalk code, the .MPP driver installs its own interrupt handlers, installs a task into the vertical retrace queue, and prepares the SCC for use. It then chooses a node ID for the Macintosh and confirms that the node ID isn't already being used by another node on the network.

Warning: For this reason it's imperative that the Macintosh be connected

to the AppleTalk network through serial port B (the printer port) before being switched on.

The AppleTalk Manager also provides Pascal routines for opening and closing the .MPP and .ATP drivers. The open calls allow a program to load AppleTalk code at times other than system startup. The close calls allow a program to remove the AppleTalk code from the Macintosh; the use of close calls is highly discouraged, since other co-resident programs are then "disconnected" from AppleTalk. Both sets of calls are described in detail under "Calling the AppleTalk Manager from Pascal".

Warning: If, at system startup, serial port B isn't available for use by AppleTalk, the .MPP driver won't open. However, a driver doesn't return an error message when it fails to open. Pascal programmers must ensure the proper opening of AppleTalk by calling one of the two routines for opening the AppleTalk drivers (either MPPOpen or ATPLoad). If AppleTalk was successfully loaded at system startup, these calls will have no effect; otherwise they'll check the availability of port B, attempt to load the AppleTalk code, and return an appropriate result code.

Assembly-language note: Assembly-language programmers can use the Pascal routines for opening AppleTalk. They can also check the availability of port B themselves and then decide whether to open MPP or ATP. Detailed information on how to do this is provided in the section "Calling the AppleTalk Manager from Assembly Language".

The two AppleTalk device drivers, named .MPP and .ATP, are included in the 128K ROM. The AppleTalk Manager, however (the interface to the drivers), is not in ROM; your application must link to the appropriate object files.

On the Macintosh Plus, you need only open the .MPP driver; this will also load the .ATP driver and NBP code automatically. Since, in the 128K ROM, device drivers return errors, it's no longer necessary to check whether port B is free and configured for AppleTalk. If port B isn't available, the .MPP driver won't open and the result code portInUse or portNotCf will be returned.

Assembly-language note: When called from assembly language, the Datagram Delivery Protocol (DDP) allows 14 (instead of 12) open sockets.

The changes to the AppleTalk manager increase functionality and resources. Two interfaces for the AppleTalk Manager calls are discussed: the new or preferred interface and the alternate interface. Picking a node address in the server range, sending packets to one's own node, multiple concurrent NBP requests, sending ATP requests through a specified socket and two new ATP calls are also discussed in this section. These calls can only be made with the preferred interface.

CALLING THE APPLETALK MANAGER FROM PASCAL

This section discusses how to use the AppleTalk Manager from Pascal. Equivalent assembly-language information is given in the "Calling the AppleTalk Manager from Assembly Language" section.

You can execute many AppleTalk Manager routines either synchronously (meaning that the application can't continue until the routine is completed) or asynchronously (meaning that the application is free to perform other tasks while the routine is being executed).

When an application calls an AppleTalk Manager routine asynchronously, an I/O request is placed in the appropriate driver's I/O queue, and control returns to the calling program—possibly even before the actual I/O is completed. Requests are taken from the queue one at a time, and processed; meanwhile, the calling program is free to work on other things.

The routines that can be executed asynchronously contain a Boolean parameter called async. If async is TRUE, the call is executed asynchronously; otherwise the call is executed synchronously. Every time an asynchronous routine call is completed, the AppleTalk Manager posts a network event. The message field of the event record will contain a handle to the parameter block that was used to make that call.

Most AppleTalk Manager routines return an integer result code of type OSErr. Each routine description lists all of the applicable result codes generated by the AppleTalk Manager, along with a short description of what the result code means. Lengthier explanations of all the result codes can be found in the summary at the end of the chapter. Result codes from other parts of the Operating System may also be returned. (See Appendix A for a list of all result codes.)

Many Pascal calls to the AppleTalk Manager require information passed in a parameter block of type ABusRecord. The exact content of an ABusRecord depends on the protocol being called:

```
TYPE  ABProtoType  =  (lapProto,ddpProto,nbpProto,atpProto);
      ABusRecord   =  RECORD
                        abOpcode:        ABCallType; {type of call}
                        abResult:        INTEGER;    {result code}
                        abUserReference: LONGINT;    {for your use}
                        CASE ABProtoType OF
                          lapProto:
                            . . .     {ALAP parameters}
                          ddpProto:
                            . . .     {DDP parameters}
                          nbpProto:
                            . . .     {NBP parameters}
                          atpProto:
                            . . .     {ATP parameters}
                        END;
```

```
                    END;

      ABRecPtr     = ^ABusRecord;
      ABRecHandle  = ^ABRecPtr;
```

The value of the abOpcode field is inserted by the AppleTalk Manager when the call is made, and is always a member of the following set:

```
TYPE  ABCallType = (tLAPRead,tLAPWrite,tDDPRead,tDDPWrite,tNBPLookup,
                    tNBPConfirm,tNBPRegister,tATPSndRequest,
                    tATPGetRequest,tATPSdRsp,tATPAddRsp,tATPRequest,
                    tATPRespond);
```

The abUserReference field is available for use by the calling program in any way it wants. This field isn't used by the AppleTalk Manager routines or drivers.

The size of an ABusRecord data structure in bytes is given by one of the following constants:

```
CONST  lapSize = 20;
       ddpSize = 26;
       nbpSize = 26;
       atpSize = 56;
```

Variables of type ABusRecord must be allocated in the heap with Memory Manager NewHandle calls. For example:

```
  myABRecord := ABRecHandle(NewHandle(ddpSize))
```

Warning: These Memory Manager calls can't be made inside interrupts.

Routines that are executed asynchronously return control to the calling program with the result code noErr as soon as the call is placed in the driver's I/O queue. This isn't an indication of successful call completion; it simply indicates that the call was successfully queued to the appropriate driver. To determine when the call is actually completed, you can either check for a network event or poll the abResult field of the call's ABusRecord. The abResult field, set to 1 when the call is made, receives the actual result code upon completion of the call.

Warning: A data structure of type ABusRecord is often used by the AppleTalk Manager during an asynchronous call, and so is locked by the AppleTalk Manager. Don't attempt to unlock or use such a variable.

Each routine description includes a list of the ABusRecord fields affected by the routine. The arrow next to each field name indicates whether it's an input, output, or input/output parameter:

Arrow	Meaning
-->	Parameter is passed to the routine
<--	Parameter is returned by the routine
<->	Parameter is passed to and returned by the routine

Opening and Closing AppleTalk

•••Click on the X-Ref button, and refer to Technical Note #224.•••

```
FUNCTION MPPOpen :  OSErr; [Not in ROM]
```

MPPOpen first checks whether the .MPP driver has already been loaded; if it has, MPPOpen does nothing and returns noErr. If MPP hasn't been loaded, MPPOpen attempts to load it into the system heap. If it succeeds, it then initializes the driver's variables and goes through the process of dynamically assigning a node ID to that Macintosh. On a Macintosh 512K or XL, it also loads the .ATP driver and NBP code into the system heap.

If serial port B isn't configured for AppleTalk, or is already in use, the .MPP driver isn't loaded and an appropriate result code is returned.

```
Result codes    noErr        No error
                portInUse    Port B is already in use
                portNotCf    Port B not configured for AppleTalk
```

```
FUNCTION MPPClose :  OSErr; [Not in ROM]
```

MPPClose removes the .MPP driver, and any data structures associated with it, from memory. If the .ATP driver or NBP code were also installed, they're removed as well. MPPClose also returns the use of port B to the Serial Driver.

Warning: Since other co-resident programs may be using AppleTalk, it's strongly recommended that you never use this call. MPPClose will completely disable AppleTalk; the only way to restore AppleTalk is to call MPPOpen again.

AppleTalk Link Access Protocol

Data Structures

ALAP calls use the following ABusRecord fields:

```
lapProto:
    (lapAddress:    LAPAdrBlock; {destination or source node ID}
     lapReqCount:   INTEGER;     {length of frame data or buffer size in bytes}
     lapActCount:   INTEGER;     {number of frame data bytes actually received}
     lapDataPtr:    Ptr);        {pointer to frame data or pointer to buffer}
```

When an ALAP frame is sent, the lapAddress field indicates the ID of the destination node. When an ALAP frame is received, lapAddress returns the ID of the source node. The lapAddress field also indicates the ALAP protocol type of the frame:

```
TYPE  LAPAdrBlock = PACKED RECORD
                      dstNodeID:    Byte;  {destination node ID}
                      srcNodeID:    Byte;  {source node ID}
                      lapProtType:  ABByte {ALAP protocol type}
                    END;
```

When an ALAP frame is sent, lapReqCount indicates the size of the frame data in bytes and lapDataPtr points to a buffer containing the frame data to be sent. When an ALAP frame is received, lapDataPtr points to a buffer in which the incoming data can be stored and lapReqCount indicates the size of the buffer in bytes. The number of bytes actually sent or received is returned in the lapActCount field.

Each ALAP frame contains an eight-bit ALAP protocol type in the header. ALAP protocol types 128 through 255 are reserved for internal use by ALAP, hence the declaration:

```
TYPE  ABByte = 1..127; {ALAP protocol type}
```

Warning: Don't use ALAP protocol type values 1 and 2; they're reserved for use by DDP. Value 3 through 15 are reserved for internal use by Apple and also shouldn't be used.

Using ALAP

Most programs will never need to call ALAP, because higher-level protocols will automatically call it as necessary. If you do want to send a frame directly via ALAP, call the LAPWrite function. If you want to read ALAP frames, you have two choices:

- Call LAPOpenProtocol with NIL for protoPtr (see below); this installs the default protocol handler provided by the AppleTalk Manager. Then call LAPRead to receive frames.
- Write your own protocol handler, and call LAPOpenProtocol to add it to the node's protocol handler table. The ALAP code will examine every incoming frame and send all those with the correct ALAP protocol type to your protocol handler. See the section "Protocol Handlers and Socket Listeners" for information on how to write a protocol handler.

When your program no longer wants to receive frames with a particular ALAP protocol type value, it can call LAPCloseProtocol to remove the corresponding protocol handler from the protocol handler table.

ALAP Routines

```
FUNCTION LAPOpenProtocol (theLAPType:  ABByte;
                          protoPtr:  Ptr) :  OSErr; [Not in ROM]
```

LAPOpenProtocol adds the ALAP protocol type specified by theLAPType to the node's protocol table. If you provide a pointer to a protocol handler in protoPtr, ALAP will send each frame with an ALAP protocol type of theLAPType to that protocol handler.

If protoPtr is NIL, the default protocol handler will be used for receiving frames with an ALAP protocol type of theLAPType. In this case, to receive a frame you must call LAPRead to provide the default protocol handler with a buffer for placing the data. If, however, you've written your own protocol handler and protoPtr points to it, your protocol handler will have the responsibility for receiving the frame and it's not necessary to call LAPRead.

```
Result codes    noErr         No error
                lapProtErr    Error attaching protocol type
```

```
FUNCTION LAPCloseProtocol (theLAPType:  ABByte) :  OSErr; [Not in ROM]
```

LAPCloseProtocol removes from the node's protocol table the specified ALAP protocol type, as well as its protocol handler.

Warning: Don't close ALAP protocol type values 1 or 2. If you close these protocol types, DDP will be disabled; once disabled, the only way to restore DDP is to restart the system, or to close and then reopen AppleTalk.

```
Result codes    noErr         No error
                lapProtErr    Error detaching protocol type
```

```
FUNCTION LAPWrite (abRecord:  ABRecHandle;
                   async:  BOOLEAN) :  OSErr; [Not in ROM]
```

```
ABusRecord
  <--    abOpcode                {always tLAPWrite}
  <--    abResult                {result code}
  -->    abUserReference         {for your use}
  -->    lapAddress.dstNodeID    {destination node ID}
  -->    lapAddress.lapProtType  {ALAP protocol type}
  -->    lapReqCount             {length of frame data}
  -->    lapDataPtr              {pointer to frame data}
```

LAPWrite sends a frame to another node. LAPReqCount and lapDataPtr specify the length and location of the data to send. The lapAddress.lapProtType field indicates the ALAP protocol type of the frame and the lapAddress.dstNodeID indicates the node ID of the node to which the frame should be sent.

Note: The first two bytes of an ALAP frame's data must contain the length in bytes of that data, including the length bytes themselves.

```
Result codes    noErr            No error
                excessCollsns    Unable to contact destination node;
                                 packet not sent
                ddpLenErr        ALAP data length too big
                lapProtErr       Invalid ALAP protocol type
```

```
FUNCTION LAPRead (abRecord:  ABRecHandle;
                  async:  BOOLEAN) :  OSErr; [Not in ROM]
```

```
ABusRecord
  <--    abOpcode                {always tLAPRead}
  <--    abResult                {result code}
  -->    abUserReference         {for your use}
  <--    lapAddress.dstNodeID    {destination node ID}
  <--    lapAddress.srcNodeID    {source node ID}
  -->    lapAddress.lapProtType  {ALAP protocol type}
  -->    lapReqCount             {buffer size in bytes}
  <--    lapActCount             {number of frame data bytes actually received}
  -->    lapDataPtr              {pointer to buffer}
```

LAPRead receives a frame from another node. LAPReqCount and lapDataPtr specify the size and location of the buffer that will receive the frame data. If the buffer isn't large enough to hold all of the incoming frame data, the extra bytes will be discarded and buf2SmallErr will be returned. The number of bytes actually received is returned in lapActCount. Only frames with ALAP protocol type equal to lapAddress.lapProtType will be received. The node IDs of the frame's source and destination nodes are returned in lapAddress.srcNodeID and lapAddress.dstNodeID. You can determine whether the packet was broadcast to you by examining the value of lapAddress.dstNodeID—if the packet was broadcast it's equal to 255, otherwise it's equal to your node ID.

Note: You should issue LAPRead calls only for ALAP protocol types that were opened (via LAPOpenProtocol) to use the default protocol handler.

Warning: If you close a protocol type for which there are still LAPRead calls pending, the calls will be canceled but the memory occupied by their ABusRecords will not be released. For this reason, before closing a protocol type, call LAPRdCancel to cancel any pending LAPRead calls associated with that protocol type.

Result codes		
	noErr	No error
	buf2SmallErr	Frame too large for buffer
	readQErr	Invalid protocol type or protocol type not found in table

```
FUNCTION LAPRdCancel (abRecord:  ABRecHandle) :  OSErr; [Not in ROM]
```

Given the handle to the ABusRecord of a previously made LAPRead call, LAPRdCancel dequeues the LAPRead call, provided that a packet satisfying the LAPRead has not already arrived. LAPRdCancel returns noErr if the LAPRead call is successfully removed from the queue. If LAPRdCancel returns recNotFnd, check the abResult field to verify that the LAPRead has been completed and determine its outcome.

Result codes		
	noErr	No error
	readQErr	Invalid protocol type or protocol type not found in table
	recNotFnd	ABRecord not found in queue

Example

This example sends an ALAP packet synchronously and waits asynchronously for a response. Assume that both nodes are using a known protocol type (in this case, 73) to receive packets, and that the destination node has a node ID of 4.

```
VAR
  myABRecord: ABRecHandle;
  myBuffer: PACKED ARRAY [0..599] OF CHAR; {buffer for both send and receive}
  myLAPType: Byte;
  errCode, index, dataLen: INTEGER;
  someText: Str255;
  async: BOOLEAN;

BEGIN
```

```
errCode := MPPOpen;
IF errCode <> noErr THEN
  WRITELN('Error in opening AppleTalk')
  {Maybe serial port B isn't available for use by AppleTalk}
ELSE
  BEGIN
  {Call Memory Manager to allocate ABusRecord}
  myABRecord := ABRecHandle(NewHandle(lapSize));
  myLAPType := 73;
 {Enter myLAPType into protocol handler table and install default handler to }
  { service frames of that ALAP type. No packets of that ALAP type will be }
  { received until we call LAPRead.}
  errCode := LAPOpenProtocol(myLAPType, NIL);
  IF errCode <> noErr THEN
    WRITELN('Error while opening the protocol type')
    {Have we opened too many protocol types? Remember that DDP uses two of }
    { them.}
  ELSE
    BEGIN
    {Prepare data to be sent}
    someText := 'This data will be in the ALAP data area';
    {The .MPP implementation requires that the first two bytes of the ALAP }
    { data field contain the length of the data, including the length bytes }
    { themselves.}
    dataLen := LENGTH(someText) + 2;
    buffer[0] := CHR(dataLen DIV 256); {high byte of data length}
    buffer[1] := CHR(dataLen MOD 256); {low byte of data length}
    FOR index := 1 TO dataLen - 2 DO {stuff buffer with packet data}
      buffer[index + 1] := someText[index];
    async := FALSE;
    WITH myABRecord^^ DO {fill parameters in the ABusRecord}
      BEGIN
      lapAddress.lapProtType := myLAPType;
      lapAddress.dstNodeID := 4;
      lapReqCount := dataLen;
      lapDataPtr := @buffer;
      END;
    {Send the frame}
    errCode := LAPWrite(myABRecord, async);
    {In the case of a sync call, errCode and the abResult field of }
    { the myABRecord will contain the same result code. We can also }
    { reuse myABRecord, since we know whether the call has completed.}
    IF errCode <> noErr THEN
      WRITELN('Error while writing out the packet')
      {Maybe the receiving node wasn't on-line}
    ELSE
      BEGIN
      {We have sent out the packet and are now waiting for a response. We }
      { issue an async LAPRead call so that we don't "hang" waiting for a }
      { response that may not come.}
      async := TRUE;
      WITH myABRecord^^ DO
        BEGIN
        lapAddress.lapProtType := myLAPType;
        {ALAP type we want to receive }
```

```
          lapReqCount := 600; {our buffer is maximum size}
          lapDataPtr := @buffer;
          END;
        errCode := LAPRead(myABRecord, async); {wait for a packet}
        IF errCode <> noErr THEN
          WRITELN('Error while trying to queue up a LAPRead')
          {Was the protocol handler installed correctly?}
        ELSE
          BEGIN
          {We can either sit here in a loop and poll the abResult }
          { field or just exit our code and use the event }
          { mechanism to flag us when the packet arrives.}
          CheckForMyEvent; {your procedure for checking for a network event}
          errCode := LAPCloseProtocol(myLAPType);
          IF errCode <> noErr THEN
            WRITELN('Error while closing the protocol type');
          END;
        END;
      END;
    END;
END.
```

Datagram Delivery Protocol

Data Structures

DDP calls use the following ABusRecord fields:

```
ddpProto:
    (ddpType:      Byte;       {DDP protocol type}
     ddpSocket:    Byte;       {source or listening socket number}
     ddpAddress:   AddrBlock;  {destination or source socket address}
     ddpReqCount:  INTEGER;    {length of datagram data or buffer size in bytes}
     ddpActCount:  INTEGER;    {number of bytes actually received}
     ddpDataPtr:   Ptr;        {pointer to buffer}
     ddpNodeID:    Byte);      {original destination node ID}
```

When a DDP datagram is sent, ddpReqCount indicates the size of the datagram data in bytes and ddpDataPtr points to a buffer containing the datagram data. DDPSocket specifies the socket from which the datagram should be sent. DDPAddress is the internet address of the socket to which the datagram should be sent:

```
TYPE  AddrBlock = PACKED RECORD
                    aNet:     INTEGER;  {network number}
                    aNode:    Byte;     {node ID}
                    aSocket:  Byte      {socket number}
                  END;
```

Note: The network number you specify in ddpAddress.aNet tells MPP whether to create a long header (for an internet) or a short header (for a local network only). A short DDP header will be sent if ddpAddress.aNet is 0 or equal to the network number of the local network.

When a DDP datagram is received, ddpDataPtr points to a buffer in which the incoming data can be stored and ddpReqCount indicates the size of the buffer in bytes. The number of bytes actually sent or received is returned in the ddpActCount field. DDPAddress is the internet address of the socket from which the datagram was sent.

DDPType is the DDP protocol type of the datagram, and ddpSocket specifies the socket that will receive the datagram.

Warning: DDP protocol types 1 through 15 and DDP socket numbers 1 through 63 are reserved by Apple for internal use. Socket numbers 64 through 127 are available for experimental use. Use of these experimental sockets isn't recommended for commercial products, since there's no mechanism for eliminating conflicting usage by different developers.

Using DDP

Before it can use a socket, the program must call DDPOpenSocket, which adds a socket and its socket listener to the socket table. When a program is finished using a socket, call DDPCloseSocket, which removes the socket's entry from the socket table. To send a datagram via DDP, call DDPWrite. To receive datagrams, you have two choices:

- Call DDPOpenSocket with NIL for sktListener (see below); this installs the default socket listener provided by the AppleTalk Manager. Then call DDPRead to receive datagrams.
- Write your own socket listener and call DDPOpenSocket to install it. DDP will call your socket listener for every incoming datagram for that socket; in this case, you shouldn't call DDPRead. For information on how to write a socket listener, see the section "Protocol Handlers and Socket Listeners".

To cancel a previously issued DDPRead call (provided it's still in the queue), call DDPRdCancel.

DDP Routines

```
FUNCTION DDPOpenSocket (VAR theSocket:  Byte;
                        sktListener:  Ptr) :  OSErr; [Not in ROM]
```

DDPOpenSocket adds a socket and its socket listener to the socket table. If theSocket is nonzero, it must be in the range 64 to 127, and it specifies the socket's number; if theSocket is 0, DDPOpenSocket dynamically assigns a socket number in the range 128 to 254, and returns it in theSocket. SktListener contains a pointer to the socket listener; if it's NIL, the default listener will be used.

If you're using the default socket listener, you must then call DDPRead to receive a datagram (in order to specify buffer space for the default socket listener). If, however, you've written your own socket listener and sktListener points to it, your listener will provide buffers for receiving datagrams and you shouldn't use DDPRead calls.

DDPOpenSocket will return ddpSktErr if you pass the number of an already opened socket, if you pass a socket number greater than 127, or if the socket table is full.

Note: The range of static socket numbers 1 through 63 is reserved by Apple for internal use. Socket numbers 64 through 127 are available for unrestricted experimental use.

```
Result codes    noErr        No error
                ddpSktErr    Socket error
```

```
FUNCTION DDPCloseSocket (theSocket:  Byte) :  OSErr; [Not in ROM]
```

DDPCloseSocket removes the entry of the specified socket from the socket table and cancels all pending DDPRead calls that have been made for that socket. If you pass a socket number of 0, or if you attempt to close a socket that isn't open, DDPCloseSocket will return ddpSktErr.

```
Result codes    noErr        No error
                ddpSktErr    Socket error
```

```
FUNCTION DDPWrite (abRecord:  ABRecHandle; doChecksum:  BOOLEAN;
                   async:  BOOLEAN) :  OSErr; [Not in ROM]
```

```
ABusRecord
  <--    abOpcode          {always tDDPWrite}
  <--    abResult          {result code}
  -->    abUserReference   {for your use}
  -->    ddpType           {DDP protocol type}
  -->    ddpSocket         {source socket number}
  -->    ddpAddress        {destination socket address}
  -->    ddpReqCount       {length of datagram data}
  -->    ddpDataPtr        {pointer to buffer}
```

DDPWrite sends a datagram to another socket. DDPReqCount and ddpDataPtr specify the length and location of the data to send. The ddpType field indicates the DDP protocol type of the frame, and ddpAddress is the complete internet address of the socket to which the datagram should be sent. DDPSocket specifies the socket from which the datagram should be sent. Datagrams sent over the internet to a node on an AppleTalk network different from the sending node's network have an optional software checksum to detect errors that might occur inside the intermediate bridges. If doChecksum is TRUE, DDPWrite will compute this checksum; if it's FALSE, this software checksum feature is ignored.

Note: The destination socket can't be in the same node as the program making the DDPWrite call.

```
Result codes    noErr          No error
                ddpLenErr      Datagram length too big
                ddpSktErr      Source socket not open
                noBridgeErr    No bridge found
```

```
FUNCTION DDPRead (abRecord:  ABRecHandle; retCksumErrs:  BOOLEAN;
                  async:  BOOLEAN) :  OSErr; [Not in ROM]
```

```
ABusRecord
  <--    abOpcode          {always tDDPRead}
  <--    abResult          {result code}
  -->    abUserReference   {for your use}
  <--    ddpType           {DDP protocol type}
  -->    ddpSocket         {listening socket number}
  <--    ddpAddress        {source socket address}
  -->    ddpReqCount       {buffer size in bytes}
  <--    ddpActCount       {number of bytes actually received}
  -->    ddpDataPtr        {pointer to buffer}
  <--    ddpNodeID         {original destination node ID}
```

DDPRead receives a datagram from another socket. The size and location of the buffer that will receive the data are specified by ddpReqCount and ddpDataPtr. If the buffer isn't large enough to hold all of the incoming frame data, the extra bytes will be discarded and buf2SmallErr will be returned. The number of bytes actually received is returned in ddpActCount. DDPSocket specifies the socket to receive the datagram (the "listening" socket). The node to which the packet was sent is returned in ddpNodeID; if the packet was broadcast ddpNodeID will contain 255. The address of the socket that sent the packet is returned in ddpAddress. If retCksumErrs is FALSE, DDPRead will discard any packets received with an invalid checksum and inform the caller of the error. If retCksumErrs is TRUE, DDPRead will deliver all packets, whether or not the checksum is valid; it will also notify the caller when there's a checksum error.

Note: The sender of the datagram must be in a different node from the receiver. You should issue DDPRead calls only for receiving datagrams for sockets opened with the default socket listener; see the description of DDPOpenSocket.

Note: If the buffer provided isn't large enough to hold all of the incoming frame data (buf2SmallErr), the checksum can't be calculated; in this case, DDPRead will deliver packets even if retCksumErrs is FALSE.

Result codes		
	noErr	No error
	buf2SmallErr	Datagram too large for buffer
	cksumErr	Checksum error
	ddpLenErr	Datagram length too big
	ddpSktErr	Socket error
	readQErr	Invalid socket or socket not found in table

```
FUNCTION DDPRdCancel (abRecord:  ABRecHandle) :  OSErr; [Not in ROM]
```

Given the handle to the ABusRecord of a previously made DDPRead call, DDPRdCancel dequeues the DDPRead call, provided that a packet satisfying the DDPRead hasn't already arrived. DDPRdCancel returns noErr if the DDPRead call is successfully removed from the queue. If DDPRdCancel returns recNotFnd, check the abResult field of abRecord to verify that the DDPRead has been completed and determine its outcome.

Result codes		
	noErr	No error
	readQErr	Invalid socket or socket not found in table
	recNotFnd	ABRecord not found in queue

Example

This example sends a DDP packet synchronously and waits asynchronously for a response. Assume that both nodes are using a known socket number (in this case, 30) to receive packets. Normally, you would want to use NBP to look up your destination's socket address.

```
VAR
  myABRecord: ABRecHandle;
  myBuffer: PACKED ARRAY [0..599] OF CHAR; {buffer for both send and receive}
  mySocket: Byte;
  errCode, index, dataLen: INTEGER;
  someText: Str255;
  async, retCksumErrs, doChecksum: BOOLEAN;

BEGIN
  errCode := MPPOpen;
  IF errCode <> noErr THEN
    WRITELN('Error in opening AppleTalk')
    {Maybe serial port B isn't available for use by AppleTalk}
  ELSE
    BEGIN
      {Call Memory Manager to allocate ABusRecord}
      myABRecord := ABRecHandle(NewHandle(ddpSize));
      mySocket := 30;
  {Add mySocket to socket table and install default socket listener to service }
{ datagrams addressed to that socket. No packets addressed to mySocket will be }
      { received until we call DDPRead. }
      errCode := DDPOpenSocket(mySocket, NIL);
      IF errCode <> noErr THEN
        WRITELN('Error while opening the socket')
      {Have we opened too many socket listeners? Remember that DDP uses two of }
        { them.}
      ELSE
        BEGIN
          {Prepare data to be sent}
          someText := 'This is a sample datagram';
          dataLen := LENGTH(someText);
          FOR index := 0 TO dataLen - 1 DO {stuff buffer with packet data}
            myBuffer[index] := someText[index + 1];
          async := FALSE;
          WITH myABRecord^^ DO {fill the parameters in the ABusRecord}
            BEGIN
              ddpType := 5;
              ddpAddress.aNet := 0; {send on "our" network}
              ddpAddress.aNode := 34;
              ddpAddress.aSocket := mySocket;
              ddpReqCount := dataLen;
              ddpDataPtr := @myBuffer;
            END;
          doChecksum := FALSE;
    {If packet contains a DDP long header, compute checksum and insert it into }
          { the header.}
          errCode := DDPWrite(myABRecord, doChecksum, async); {send packet}
     {In the case of a sync call, errCode and the abResult field of myABRecord }
   { will contain the same result code. We can also reuse myABRecord, since we }
```

```
          { know whether the call has completed.}
          IF errCode <> noErr THEN
            WRITELN('Error while writing out the packet')
            {Maybe the receiving node wasn't on-line}
          ELSE
            BEGIN
           {We have sent out the packet and are now waiting for a response. We }
           { issue an async DDPRead call so that we don't "hang" waiting for a }
          { response that may not come. To cancel the async read call, we must }
              { close the socket associated with the call or call DDPRdCancel.}
              async := TRUE;
              retCksumErrs := TRUE; {return packets even if }
                                    { they have a checksum error}
              WITH myABRecord^^ DO
                BEGIN
                  ddpSocket := mySocket;
                  ddpReqCount := 600; {our reception buffer is max size}
                  ddpDataPtr := @myBuffer;
                END;
              {Wait for a packet asynchronously}
              errCode := DDPRead(myABRecord, retCksumErrs, async);
              IF errCode <> noErr THEN
                WRITELN('Error while trying to queue up a DDPRead')
                {Was the socket listener installed correctly?}
              ELSE
                BEGIN
                  {We can either sit here in a loop and poll the }
                  { abResult field or just exit our code and use the }
                  { event mechanism to flag us when the packet arrives.}
                  CheckForMyEvent; {your procedure for checking for a }
                                   { network event}
                  {If there were no errors, a packet is inside the array }
                  { mybuffer, the length is in ddpActCount, and the }
                  { address of the sending socket is in ddpAddress. }
                  { Process the packet received here and report any errors.}
                  errCode := DDPCloseSocket(mySocket); {we're done with it}
                  IF errCode <> noErr THEN
                    WRITELN('Error while closing the socket');
                END;
            END;
        END;
    END;
END.
```

AppleTalk Transaction Protocol

Data Structures

ATP calls use the following ABusRecord fields:

```
atpProto:
  (atpSocket:     Byte;        {listening or responding socket number}
   atpAddress:    AddrBlock;   {destination or source socket address}
```

```
   atpReqCount:   INTEGER;     {request size or buffer size}
   atpDataPtr:    Ptr;         {pointer to buffer}
   atpRspBDSPtr:  BDSPtr;      {pointer to response BDS}
   atpBitMap:     BitMapType;  {transaction bit map}
   atpTransID:    INTEGER;     {transaction ID}
   atpActCount:   INTEGER;     {number of bytes actually received}
   atpUserData:   LONGINT;     {user bytes}
   atpXO:         BOOLEAN;     {exactly-once flag}
   atpEOM:        BOOLEAN;     {end-of-message flag}
   atpTimeOut:    Byte;        {retry timeout interval in seconds}
   atpRetries:    Byte;        {maximum number of retries}
   atpNumBufs:    Byte;        {number of elements in response BDS or number }
                               { of response packets sent}
   atpNumRsp:     Byte;        {number of response packets received or }
                               { sequence number}
   atpBDSSize:    Byte;        {number of elements in response BDS}
   atpRspUData:   LONGINT;     {user bytes sent or received in transaction }
                               { response}
   atpRspBuf:     Ptr;         {pointer to response message buffer}
   atpRspSize:    INTEGER);    {size of response message buffer}
```

The socket receiving the request or sending the response is identified by atpSocket. ATPAddress is the address of either the destination or the source socket of a transaction, depending on whether the call is sending or receiving data, respectively. ATPDataPtr and atpReqCount specify the location and size (in bytes) of a buffer that either contains a request or will receive a request. The number of bytes actually received in a request is returned in atpActCount. ATPTransID specifies the transaction ID. The transaction bit map is contained in atpBitMap, in the form:

```
TYPE BitMapType = PACKED ARRAY[0..7] OF BOOLEAN;
```

Each bit in the bit map corresponds to one of the eight possible packets in a response. For example, when a request is made for which five response packets are expected, the bit map sent is binary 00011111 or decimal 31. If the second packet in the response is lost, the requesting socket will retransmit the request with a bit map of binary 00000010 or decimal 2.

ATPUserData contains the user bytes of an ATP header. ATPXO is TRUE if the transaction is to be made with exactly-once service. ATPEOM is TRUE if the response packet is the last packet of a transaction. If the number of responses is less than the number that were requested, then ATPEOM must also be TRUE. ATPNumRsp contains either the number of responses received or the sequence number of a response.

The timeout interval in seconds and the maximum number of times that a request should be made are indicated by atpTimeOut and atpRetries, respectively.

Note: Setting atpRetries to 255 will cause the request to be retransmitted indefinitely, until a full response is received or the call is canceled.

ATP provides a data structure, known as a response buffer data structure (response BDS), for allocating buffer space to receive the datagram(s) of the response. A response BDS is an array of one to eight elements. Each BDS element defines the size and location of a buffer for receiving one response datagram;

they're numbered 0 to 7 to correspond to the sequence numbers of the response datagrams.

ATP needs a separate buffer for each response datagram expected, since packets may not arrive in the proper sequence. It does not, however, require you to set up and use the BDS data structure to describe the response buffers; if you don't, ATP will do it for you. Two sets of calls are provided for both requests and responses; one set requires you to allocate a response BDS and the other doesn't.

Assembly-language note: The two calls that don't require you to define a BDS data structure (ATPRequest and ATPResponse) are available in Pascal only.

The number of BDS elements allocated (in other words, the maximum number of datagrams that the responder can send) is indicated in the TReq by an eight-bit bit map. The bits of this bit map are numbered 0 to 7 (the least significant bit being number 0); each bit corresponds to the response datagram with the respective sequence number.

ATPRspBDSPtr and atpBDSSize indicate the location and number of elements in the response BDS, which has the following structure:

```
TYPE  BDSElement =
            RECORD
              buffSize:   INTEGER;  {buffer size in bytes}
              buffPtr:    Ptr;      {pointer to buffer}
              dataSize:   INTEGER;  {number of bytes actually received}
              userBytes:  LONGINT   {user bytes}
            END;

      BDSType = ARRAY[0..7] OF BDSElement; {response BDS}
      BDSPtr  = ^BDSType;
```

ATPNumBufs indicates the number of elements in the response BDS that contain information. In most cases, you can allocate space for your variables of BDSType statically with a VAR declaration. However, you can allocate only the minimum space required by your ATP calls by doing the following:

```
  VAR myBDSPtr:  BDSPtr;
      . . .
  numOfBDS := 3; {number of elements needed}
  myBDSPtr := BDSPtr(NewPtr(SIZEOF(BDSElement) * numOfBDS));
```

Note: The userBytes field of the BDSElement and the atpUserData field of the ABusRecord represent the same information in the datagram. Depending on the ATP call made, one or both of these fields will be used.

Using ATP

Before you can use ATP on a Macintosh 128K, the .ATP driver must be read from the system resource file via an ATPLoad call. The .ATP driver loads itself into the application heap and installs a task into the vertical retrace queue.

Warning: When another application starts up, the application heap is

reinitialized; on a Macintosh 128K, this means that the ATP code is lost (and must be reloaded by the next application).

When you're through using ATP on a Macintosh 128K, call ATPUnload—the system will be returned to the state it was in before the .ATP driver was opened.

On a Macintosh 512K or XL, the .ATP driver will have been loaded into the system heap either at system startup or upon execution of MPPOpen or ATPLoad. ATPUnload has no effect on a Macintosh 512K or XL.

To send a transaction request, call ATPSndRequest or ATPRequest. The .ATP driver will automatically select and open a socket through which the request datagram will be sent, and through which the response datagrams will be received. The requester must specify the full network address (network number, node ID, and socket number) of the socket to which the request is to be sent. This socket is known as the responding socket, and its address must be known in advance by the requester.

At the responder's end, before a transaction request can be received, a responding socket must be opened, and the appropriate calls be made, to receive a request. To do this, the responder first makes an ATPOpenSocket call which allows the responder to specify the address (or part of it) of the requesters from whom it's willing to accept transaction requests. Then it issues an ATPGetRequest call to provide ATP with a buffer for receiving a request; when a request is received, ATPGetRequest is completed. The responder can queue up several ATPGetRequest calls, each of which will be completed as requests are received.

Upon receiving a request, the responder performs the requested operation, and then prepares the information to be returned to the requester. It then calls ATPSndRsp (or ATPResponse) to send the response. Actually, the responder can issue the ATPSndRsp call with only part (or none) of the response specified. Additional portions of the response can be sent later by calling ATPAddRsp.

The ATPSndRsp and ATPAddRsp calls provide flexibility in the design (and range of types) of transaction responders. For instance, the responder may, for some reason, be forced to send the responses out of sequence. Also, there might be memory constraints that force sending the complete transaction response in parts. Even though eight response datagrams might need to be sent, the responder might have only enough memory to build one datagram at a time. In this case, it would build the first response datagram and call ATPSndRsp to send it. It would then build the second response datagram in the same buffer and call ATPAddRsp to send it; and so on, for the third through eighth response datagrams.

A responder can close a responding socket by calling ATPCloseSocket. This call cancels all pending ATP calls for that socket, such as ATPGetRequest, ATPSndRsp, and ATPResponse.

For exactly-once transactions, the ATPSndRsp and ATPAddRsp calls don't terminate until the entire transaction has completed (that is, the responding end receives a release packet, or the release timer has expired).

To cancel a pending, asynchronous ATPSndRequest or ATPRequest call, call ATPReqCancel. To cancel a pending, asynchronous ATPSndRsp or ATPResponse call, call ATPRspCancel. Pending asynchronous ATPGetRequest calls can be canceled only

by issuing the ATPCloseSocket call, but that will cancel all outstanding calls for that socket.

•••Click on the X-Ref button, and refer to Technical Note #250.•••

Warning: You cannot reuse a variable of type ABusRecord passed to an ATP routine until the entire transaction has either been completed or canceled.

ATP Routines

```
FUNCTION ATPLoad :  OSErr; [Not in ROM]
```

•••Click on the X-Ref button, and refer to Technical Note #224.•••

ATPLoad first verifies that the .MPP driver is loaded and running. If it isn't, ATPLoad verifies that port B is configured for AppleTalk and isn't in use, and then loads MPP into the system heap.

ATPLoad then loads the .ATP driver, unless it's already in memory. On a Macintosh 128K, ATPLoad reads the .ATP driver from the system resource file into the application heap; on a Macintosh 512K or XL, ATP is read into the system heap.

Note: On a Macintosh 512K or XL, ATPLoad and MPPOpen perform essentially the same function.

```
Result codes    noErr        No error
                portInUse    Port B is already in use
                portNotCf    Port B not configured for AppleTalk
```

```
FUNCTION ATPUnload :  OSErr; [Not in ROM]
```

ATPUnload makes the .ATP driver purgeable; the space isn't actually released by the Memory Manager until necessary.

Note: This call applies only to a Macintosh 128K; on a Macintosh 512K or Macintosh XL, ATPUnload has no effect.

```
Result codes    noErr    No error
```

```
FUNCTION ATPOpenSocket (addrRcvd:  AddrBlock;
                        VAR atpSocket:  Byte) :  OSErr; [Not in ROM]
```

ATPOpenSocket opens a socket for the purpose of receiving requests. ATPSocket contains the socket number of the socket to open; if it's 0, a number is dynamically assigned and returned in atpSocket. AddrRcvd contains a filter of the sockets from which requests will be accepted. A 0 in the network number, node ID, or socket number field of the addrRcvd record acts as a "wild card"; for instance, a 0 in the socket number field means that requests will be accepted from all sockets in the node(s) specified by the network and node fields.

```
Result codes    noErr          No error
                tooManySkts    Socket table full
```

```
                noDataArea     Too many outstanding ATP calls
```

Note: If you're only going to send requests and receive responses to these requests, you don't need to open an ATP socket. When you make the ATPSndRequest or ATPRequest call, ATP automatically opens a dynamically assigned socket for that purpose.

```
FUNCTION ATPCloseSocket (atpSocket:  Byte) :  OSErr; [Not in ROM]
```

ATPCloseSocket closes the responding socket whose number is specified by atpSocket. It releases the data structures associated with all pending, asynchronous calls involving that socket; these pending calls are completed immediately and return the result code sktClosed.

```
Result codes    noErr         No error
                noDataArea    Too many outstanding ATP calls
```

```
FUNCTION ATPSndRequest (abRecord:  ABRecHandle;
                        async:  BOOLEAN) :  OSErr; [Not in ROM]
```

```
ABusRecord
  <--    abOpcode          {always tATPSndRequest}
  <--    abResult          {result code}
  -->    abUserReference   {for your use}
  -->    atpAddress        {destination socket address}
  -->    atpReqCount       {request size in bytes}
  -->    atpDataPtr        {pointer to buffer}
  -->    atpRspBDSPtr      {pointer to response BDS}
  -->    atpUserData       {user bytes}
  -->    atpXO             {exactly-once flag}
  <--    atpEOM            {end-of-message flag}
  -->    atpTimeOut        {retry timeout interval in seconds}
  -->    atpRetries        {maximum number of retries}
  -->    atpNumBufs        {number of elements in response BDS}
  <--    atpNumRsp         {number of response packets actually received}
```

ATPSndRequest sends a request to another socket. ATPAddress is the internet address of the socket to which the request should be sent. ATPDataPtr and atpReqCount specify the location and size of a buffer that contains the request information to be sent. ATPUserData contains the user bytes for the ATP header.

ATPSndRequest requires you to allocate a response BDS. ATPRspBDSPtr is a pointer to the response BDS; atpNumBufs indicates the number of elements in the BDS (this is also the maximum number of response datagrams that will be accepted). The number of response datagrams actually received is returned in atpNumRsp; if a nonzero value is returned, you can examine the response BDS to determine which packets of the transaction were actually received. If the number returned is less than requested, one of the following is true:

- Some of the packets have been lost and the retry count has been exceeded.
- ATPEOM is TRUE; this means that the response consisted of fewer packets than were expected, but that all packets sent were received (the last packet came with the atpEOM flag set).

ATPTimeOut indicates the length of time that ATPSndRequest should wait for a response before retransmitting the request. ATPRetries indicates the maximum number of retries ATPSndRequest should attempt. ATPXO should be TRUE if you want the request to be part of an exactly-once transaction.

ATPSndRequest completes when either the transaction is completed or the retry count is exceeded.

```
Result codes    noErr          No error
                reqFailed      Retry count exceeded
                tooManyReqs    Too many concurrent requests
                noDataArea     Too many outstanding ATP callsFUNCTION ATPRequest
(abRecord:  ABRecHandle;
                    async:  BOOLEAN) :  OSErr; [Not in ROM]
```

```
ABusRecord
  <--    abOpcode          {always tATPRequest}
  <--    abResult          {result code}
  -->    abUserReference   {for your use}
  -->    atpAddress        {destination socket address}
  -->    atpReqCount       {request size in bytes}
  -->    atpDataPtr        {pointer to buffer}
  <--    atpActCount       {number of bytes actually received}
  -->    atpUserData       {user bytes}
  -->    atpXO             {exactly-once flag}
  <--    atpEOM            {end-of-message flag}
  -->    atpTimeOut        {retry timeout interval in seconds}
  -->    atpRetries        {maximum number of retries}
  <--    atpRspUData       {user bytes received in transaction response}
  -->    atpRspBuf         {pointer to response message buffer}
  -->    atpRspSize        {size of response message buffer}
```

ATPRequest is functionally analogous to ATPSndRequest. It sends a request to another socket, but doesn't require the caller to set up and use the BDS data structure to describe the response buffers. ATPAddress indicates the socket to which the request should be sent. ATPDataPtr and atpReqCount specify the location and size of a buffer that contains the request information to be sent. ATPUserData contains the user bytes to be sent in the request's ATP header. ATPTimeOut indicates the length of time that ATPRequest should wait for a response before retransmitting the request. ATPRetries indicates the maximum number of retries ATPRequest should attempt.

To use this call, you must have an area of contiguous buffer space that's large enough to receive all expected datagrams. The various datagrams will be assembled in this buffer and returned to you as a complete message upon completion of the transaction. The location and size of this buffer are passed in atpRspBuf and atpRspSize. Upon completion of the call, the size of the received response message is returned in atpActCount. The user bytes received in the ATP header of the first response packet are returned in atpRspUData. ATPXO should be TRUE if you want the request to be part of an exactly-once transaction.

Although you don't provide a BDS, ATPRequest in fact creates one and calls the

.ATP driver (as in an ATPSndRequest call). For this reason, the abRecord fields atpRspBDSPtr and atpNumBufs are used by ATPRequest; you should not expect these fields to remain unaltered during or after the function's execution.

For ATPRequest to receive and correctly deliver the response as a single message, the responding end must, upon receiving the request (with an ATPGetRequest call), generate the complete response as a message in a single buffer and then call ATPResponse.

Note: The responding end could also use ATPSndRsp and ATPAddRsp provided that each response packet (except the last one) contains exactly 578 ATP data bytes; the last packet in the response can contain less than 578 ATP data bytes. Also, if this method is used, only the ATP user bytes of the first response packet will be delivered to the requester; any information in the user bytes of the remaining response packets will not be delivered.

ATPRequest completes when either the transaction is completed or the retry count is exceeded.

Result codes		
	noErr	No error
	reqFailed	Retry count exceeded
	tooManyReqs	Too many concurrent requests
	sktClosed	Socket closed by a cancel call
	noDataArea	Too many outstanding ATP calls

```
FUNCTION ATPReqCancel (abRecord:  ABRecHandle;
                       async:  BOOLEAN) :  OSErr; [Not in ROM]
```

Given the handle to the ABusRecord of a previously made ATPSndRequest or ATPRequest call, ATPReqCancel dequeues the ATPSndRequest or ATPRequest call, provided that the call hasn't already completed. ATPReqCancel returns noErr if the ATPSndRequest or ATPRequest call is successfully removed from the queue. If it returns cbNotFound, check the abResult field of abRecord to verify that the ATPSndRequest or ATPRequest call has completed and determine its outcome.

Result codes		
	noErr	No error
	cbNotFound	ATP control block not found

```
FUNCTION ATPGetRequest (abRecord:  ABRecHandle;
                        async:  BOOLEAN) :  OSErr; [Not in ROM]
```

```
ABusRecord
  <--    abOpcode         {always tATPGetRequest}
  <--    abResult         {result code}
  -->    abUserReference  {for your use}
  -->    atpSocket        {listening socket number}
  <--    atpAddress       {source socket address}
  -->    atpReqCount      {buffer size in bytes}
  -->    atpDataPtr       {pointer to buffer}
  <--    atpBitMap        {transaction bit map}
  <--    atpTransID       {transaction ID}
  <--    atpActCount      {number of bytes actually received}
  <--    atpUserData      {user bytes}
  <--    atpXO            {exactly-once flag}
```

ATPGetRequest sets up the mechanism to receive a request sent by either an ATPSndRequest or an ATPRequest call. ATPSocket contains the socket number of the socket that should listen for a request; this socket must already have been opened by calling ATPOpenSocket. The address of the socket from which the request was sent is returned in atpAddress. ATPDataPtr specifies a buffer to store the incoming request; atpReqCount indicates the size of the buffer in bytes. The number of bytes actually received in the request is returned in atpActCount. ATPUserData contains the user bytes from the ATP header. The transaction bit map is returned in atpBitMap. The transaction ID is returned in atpTransID. ATPXO will be TRUE if the request is part of an exactly-once transaction.

ATPGetRequest completes when a request is received. To cancel an asynchronous ATPGetRequest call, you must call ATPCloseSocket, but this cancels all pending calls involving that socket.

```
Result codes    noErr        No error
                badATPSkt    Bad responding socket
                sktClosed    Socket closed by a cancel call
```

```
FUNCTION ATPSndRsp (abRecord:  ABRecHandle;
                    async:  BOOLEAN) :  OSErr; [Not in ROM]
```

```
ABusRecord
  <--    abOpcode          {always tATPSdRsp}
  <--    abResult          {result code}
  -->    abUserReference   {for your use}
  -->    atpSocket         {responding socket number}
  -->    atpAddress        {destination socket address}
  -->    atpRspBDSPtr      {pointer to response BDS}
  -->    atpTransID        {transaction ID}
  -->    atpEOM            {end-of-message flag}
  -->    atpNumBufs        {number of response packets being sent}
  -->    atpBDSSize        {number of elements in response BDS}
```

ATPSndRsp sends a response to another socket. ATPSocket contains the socket number from which the response should be sent and atpAddress contains the internet address of the socket to which the response should be sent. ATPTransID must contain the transaction ID. ATPEOM is TRUE if the response BDS contains the final packet in a transaction composed of a group of packets and the number of packets in the response is less than expected. ATPRspBDSPtr points to the buffer data structure containing the responses to be sent. ATPBDSSize indicates the number of elements in the response BDS, and must be in the range 1 to 8. ATPNumBufs indicates the number of response packets being sent with this call, and must be in the range 0 to 8.

Note: In some situations, you may want to send only part (or possibly none) of your response message back immediately. For instance, you might be requested to send back seven disk blocks, but have only enough internal memory to store one block. In this case, set atpBDSSize to 7 (total number of response packets), atpNumBufs to 0 (number of response packets currently being sent), and call ATPSndRsp. Then as you read in one block at a time, call ATPAddRsp until all seven response datagrams have been sent.

During exactly-once transactions, ATPSndRsp won't complete until the release packet is received or the release timer expires.

```
Result codes    noErr         No error
                badATPSkt     Bad responding socket
                noRelErr      No release received
                sktClosed     Socket closed by a cancel call
                noDataArea    Too many outstanding ATP calls
                badBuffNum    Bad sequence number
```

```
FUNCTION ATPAddRsp (abRecord:  ABRecHandle) :  OSErr; [Not in ROM]
```

```
ABusRecord
  <--    abOpcode         {always tATPAddRsp}
  <--    abResult         {result code}
  -->    abUserReference  {for your use}
  -->    atpSocket        {responding socket number}
  -->    atpAddress       {destination socket address}
  -->    atpReqCount      {buffer size in bytes}
  -->    atpDataPtr       {pointer to buffer}
  -->    atpTransID       {transaction ID}
  -->    atpUserData      {user bytes}
  -->    atpEOM           {end-of-message flag}
  -->    atpNumRsp        {sequence number}
```

ATPAddRsp sends one additional response packet to a socket that has already been sent the initial part of a response via ATPSndRsp. ATPSocket contains the socket number from which the response should be sent and atpAddress contains the internet address of the socket to which the response should be sent. ATPTransID must contain the transaction ID. ATPDataPtr and atpReqCount specify the location and size of a buffer that contains the information to send; atpNumRsp is the sequence number of the response. ATPEOM is TRUE if this response datagram is the final packet in a transaction composed of a group of packets. ATPUserData contains the user bytes to be sent in this response datagram's ATP header.

Note: No BDS is needed with ATPAddRsp because all pertinent information is passed within the record.

```
Result codes    noErr         No error
                badATPSkt     Bad responding socket
                badBuffNum    Bad sequence number
                noSendResp    ATPAddRsp issued before ATPSndRsp
                noDataArea    Too many outstanding ATP calls
```

```
FUNCTION ATPResponse (abRecord:  ABRecHandle;
                      async:  BOOLEAN) :  OSErr; [Not in ROM]
```

```
ABusRecord
  <--    abOpcode         {always tATPResponse}
  <--    abResult         {result code}
  -->    abUserReference  {for your use}
  -->    atpSocket        {responding socket number}
  -->    atpAddress       {destination socket address}
  -->    atpTransID       {transaction ID)
```

```
  -->    atpRspUData      {user bytes sent in transaction response}
  -->    atpRspBuf        {pointer to response message buffer}
  -->    atpRspSize       {size of response message buffer}
```

ATPResponse is functionally analogous to ATPSndRsp. It sends a response to another socket, but doesn't require the caller to provide a BDS. ATPAddress must contain the complete network address of the socket to which the response should be sent (taken from the data provided by an ATPGetRequest call). ATPTransID must contain the transaction ID. ATPSocket indicates the socket from which the response should be sent (the socket on which the corresponding ATPGetRequest was issued). ATPRspBuf points to the buffer containing the response message; the size of this buffer must be passed in atpRspSize. The four user bytes to be sent in the ATP header of the first response packet are passed in atpRspUData. The last packet of the transaction response is sent with the EOM flag set.

Although you don't provide a BDS, ATPResponse in fact creates one and calls the .ATP driver (as in an ATPSndRsp call). For this reason, the abRecord fields atpRspBDSPtr and atpNumBufs are used by ATPResponse; you should not expect these fields to remain unaltered during or after the function's execution.

During exactly-once transactions ATPResponse won't complete until the release packet is received or the release timer expires.

Warning: The maximum permissible size of the response message is 4624 bytes.

```
Result codes    noErr         No error
                badATPSkt     Bad responding socket
                noRelErr      No release received
                atpLenErr     Response too big
                sktClosed     Socket closed by a cancel call
                noDataArea    Too many outstanding ATP calls
```

```
FUNCTION ATPRspCancel (abRecord:  ABRecHandle;
                       async:  BOOLEAN) :  OSErr; [Not in ROM]
```

Given the handle to the ABusRecord of a previously made ATPSndRsp or ATPResponse call, ATPRspCancel dequeues the ATPSndRsp or ATPResponse call, provided that the call hasn't already completed. ATPRspCancel returns noErr if the ATPSndRsp or ATPResponse call is successfully removed from the queue. If it returns cbNotFound, check the abResult field of abRecord to verify that the ATPSndRsp or ATPResponse call has completed and determine its outcome.

```
Result codes    noErr         No error
                cbNotFound    ATP control block not found
```

Example

This example shows the requesting side of an ATP transaction that asks for a 512-byte disk block from the responding end. The block number of the file is a byte and is contained in myBuffer[0].

```
VAR
  myABRecord: ABRecHandle;
  myBDSPtr: BDSPtr;
  myBuffer: PACKED ARRAY [0..511] OF CHAR;
```

```
  errCode: INTEGER;
  async: BOOLEAN;

BEGIN
  errCode := ATPLoad;
  IF errCode <> noErr THEN
    WRITELN('Error in opening AppleTalk')
    {Maybe serial port B isn't available for use by AppleTalk}
  ELSE
    BEGIN
      {Prepare the BDS; allocate space for a one-element BDS}
      myBDSPtr := BDSPtr(NewPtr(SIZEOF(BDSElement)));
      WITH myBDSPtr^[0] DO
        BEGIN
          buffSize := 512; {size of our buffer used in reception}
          buffPtr := @myBuffer; {pointer to the buffer}
        END;
      {Prepare the ABusRecord}
      myBuffer[0] := CHR(1); {requesting disk block number 1}
      myABRecord := ABRecHandle(NewHandle(atpSize));
      WITH myABRecord^^ DO
        BEGIN
          atpAddress.aNet := 0;
          atpAddress.aNode := 30; {we probably got this from an NBP call}
          atpAddress.aSocket := 15; {socket to send request to}
          atpReqCount := 1; {size of request data field (disk block #)}
          atpDataPtr := @myBuffer; {ptr to request to be sent}
          atpRspBDSPtr := @myBDSPtr;
          atpUserData := 0; {for your use}
          atpXO := FALSE; {at-least-once service}
          atpTimeOut := 5; {5-second timeout}
          atpRetries := 3; {3 retries; request will be sent 4 times max}
          atpNumBufs := 1; {we're only expecting 1 block to be returned}
        END;
      async := FALSE;
      {Send the request and wait for the response}
      errCode := ATPSndRequest(myABRecord, async);
      IF errCode <> noErr THEN
        WRITELN('An error occurred in the ATPSndRequest call')
      ELSE
        BEGIN
          {The disk block requested is now in myBuffer. We can verify }
          { that atpNumRsp contains 1, meaning one response received.}
        . . .
        END;
    END;
END.
```

Name-Binding Protocol

Data Structures

NBP calls use the following fields:

```
nbpProto:
  (nbpEntityPtr:       EntityPtr;     {pointer to entity name}
   nbpBufPtr:          Ptr;           {pointer to buffer}
   nbpBufSize:         INTEGER;       {buffer size in bytes}
   nbpDataField:       INTEGER;       {number of addresses or socket number}
   nbpAddress:         AddrBlock;     {socket address}
   nbpRetransmitInfo:  RetransType);  {retransmission information}
```

When data is sent via NBP, nbpBufSize indicates the size of the data in bytes and nbpBufPtr points to a buffer containing the data. When data is received via NBP, nbpBufPtr points to a buffer in which the incoming data can be stored and nbpBufSize indicates the size of the buffer in bytes. NBPAddress is used in some calls to give the internet address of a named entity. The AddrBlock data type is described above under "Datagram Delivery Protocol".

NBPEntityPtr points to a variable of type EntityName, which has the following data structure:

```
TYPE  EntityName = RECORD
                     objStr:   Str32;   {object}
                     typeStr:  Str32;   {type}
                     zoneStr:  Str32    {zone}
                   END;

      EntityPtr = ^EntityName;
      Str32 = STRING[32];
```

NBPRetransmitInfo contains information about the number of times a packet should be transmitted and the interval between retransmissions:

```
TYPE  RetransType =  PACKED RECORD
                       retransInterval:  Byte;    {retransmit interval in }
                                                  { 8-tick units}
                       retransCount:     Byte     {total number of attempts}
                     END;
```

RetransCount contains the total number of times a packet should be transmitted, including the first transmission. If retransCount is 0, the packet will be transmitted a total of 255 times.

Using NBP

On a Macintosh 128K, the AppleTalk Manager's NBP code is read into the application heap when any one of the NBP (Pascal) routines is called; you can call the NBPLoad function yourself if you want to load the NBP code explicitly. When you're finished with the NBP code and want to reclaim the space it occupies, call NBPUnload. On a Macintosh 512K or XL, the NBP code is read in when the .MPP driver is loaded.

Note: When another application starts up, the application heap is reinitialized; on a Macintosh 128K, this means that the NBP code is lost (and must be reloaded by the next application).

When an entity wants to communicate via an AppleTalk network, it should call NBPRegister to place its name and internet address in the names table. When an entity no longer wants to communicate on the network, or is being shut down, it should call NBPRemove to remove its entry from the names table.

To determine the address of an entity you know only by name, call NBPLookup, which returns a list of all entities with the name you specify. Call NBPExtract to extract entity names from the list.

If you already know the address of an entity, and want only to confirm that it still exists, call NBPConfirm. NBPConfirm is more efficient than NBPLookup in terms of network traffic.

NBP Routines

```
FUNCTION NBPRegister (abRecord:  ABRecHandle;
                      async:  BOOLEAN) :  OSErr; [Not in ROM]

ABusRecord
  <--    abOpcode            {always tNBPRegister}
  <--    abResult            {result code}
  -->    abUserReference     {for your use}
  -->    nbpEntityPtr        {pointer to entity name}
  -->    nbpBufPtr           {pointer to buffer}
  -->    nbpBufSize          {buffer size in bytes}
  -->    nbpAddress.aSocket  {socket address}
  -->    nbpRetransmitInfo   {retransmission information}
```

NBPRegister adds the name and address of an entity to the node's names table. NBPEntityPtr points to a variable of type EntityName containing the entity's name. If the name is already registered, NBPRegister returns the result code nbpDuplicate. NBPAddress indicates the socket for which the name should be registered. NBPBufPtr and nbpBufSize specify the location and size of a buffer for NBP to use internally.

While the variable of type EntityName is declared as three 32-byte strings, only the actual characters of the name are placed in the buffer pointed to by nbpBufPtr. For this reason, nbpBufSize needs only to be equal to the actual length of the name, plus an additional 12 bytes for use by NBP.

Warning: This buffer must not be altered or released until the name is removed from the names table via an NBPRemove call. If you allocate the buffer through a NewHandle call, you must lock it as long as the name is registered.

Warning: The zone field of the entity name must be set to the meta-character "*".

```
Result codes    noErr           No error
                nbpDuplicate    Duplicate name already exists

FUNCTION NBPLookup (abRecord:  ABRecHandle;
                    async:  BOOLEAN) :  OSErr; [Not in ROM]

ABusRecord
```

```
  <--    abOpcode           {always tNBPLookup}
  <--    abResult           {result code}
  -->    abUserReference    {for your use}
  -->    nbpEntityPtr       {pointer to entity name}
  -->    nbpBufPtr          {pointer to buffer}
  -->    nbpBufSize         {buffer size in bytes}
  <->    nbpDataField       {number of addresses received}
  -->    nbpRetransmitInfo  {retransmission information}
```

NBPLookup returns the addresses of all entities with a specified name. NBPEntityPtr points to a variable of type EntityName containing the name of the entity whose address should be returned. (Meta-characters are allowed in the entity name.) NBPBufPtr and nbpBufSize contain the location and size of an area of memory in which the entity names and their corresponding addresses should be returned. NBPDataField indicates the maximum number of matching names to find addresses for; the actual number of addresses found is returned in nbpDataField. NBPRetransmitInfo contains the retry interval and the retry count.

When specifying nbpBufSize, for each NBP tuple expected, allow space for the actual characters of the name, the address, and four bytes for use by NBP.

```
Result codes    noErr         No error
                nbpBuffOvr    Buffer overflow
```

```
FUNCTION NBPExtract (theBuffer:  Ptr; numInBuf:  INTEGER; whichOne:  INTEGER;
                     VAR abEntity:  EntityName;
                     VAR address:  AddrBlock) :  OSErr; [Not in ROM]
```

NBPExtract returns one address from the list of addresses returned by NBPLookup. TheBuffer and numInBuf indicate the location and number of tuples in the buffer. WhichOne specifies which one of the tuples in the buffer should be returned in the abEntity and address parameters.

```
Result codes    noErr         No error
                extractErr    Can't find tuple in buffer
```

```
FUNCTION NBPConfirm (abRecord:  ABRecHandle;
                     async:  BOOLEAN) :  OSErr; [Not in ROM]
```

```
ABusRecord
  <--    abOpcode           {always tNBPConfirm}
  <--    abResult           {result code}
  -->    abUserReference    {for your use}
  -->    nbpEntityPtr       {pointer to entity name}
  <--    nbpDataField       {socket number}
  -->    nbpAddress         {socket address}
  -->    nbpRetransmitInfo  {retransmission information}
```

NBPConfirm confirms that an entity known by name and address still exists (is still entered in the names directory). NBPEntityPtr points to a variable of type EntityName that contains the name to confirm, and nbpAddress specifies the address to be confirmed. (No meta-characters are allowed in the entity name.) NBPRetransmitInfo contains the retry interval and the retry count. The socket number of the entity is returned in nbpDataField. NBPConfirm is more efficient than NBPLookup in terms of network traffic.

```
Result codes    noErr           No error
                nbpConfDiff     Name confirmed for different socket
                nbpNoConfirm    Name not confirmed
```

```
FUNCTION NBPRemove (abEntity:  EntityPtr) :  OSErr; [Not in ROM]
```

NBPRemove removes an entity name from the names table of the given entity's node.

```
Result codes    noErr          No error
                nbpNotFound    Name not found
```

```
FUNCTION NBPLoad :  OSErr; [Not in ROM]
```

On a Macintosh 128K, NBPLoad reads the NBP code from the system resource file into the application heap. On a Macintosh 512K or XL, NBPLoad has no effect since the NBP code should have already been loaded when the .MPP driver was opened. Normally you'll never need to call NBPLoad, because the AppleTalk Manager calls it when necessary.

```
Result codes    noErr    No error
```

```
FUNCTION NBPUnload :  OSErr; [Not in ROM]
```

On a Macintosh 128K, NBPUnload makes the NBP code purgeable; the space isn't actually released by the Memory Manager until necessary. On a Macintosh 512K or Macintosh XL, NBPUnload has no effect.

```
Result codes    noErr    No error
```

Example

This example of NBP registers our node as a print spooler, searches for any print spoolers registered on the network, and then extracts the information for the first one found.

```
CONST
  mySocket = 20;

VAR
  myABRecord: ABRecHandle;
  myEntity: EntityName;
  entityAddr: AddrBlock;
  nbpNamePtr: Ptr;
  myBuffer: PACKED ARRAY [0..999] OF CHAR;
  errCode: INTEGER;
  async: BOOLEAN;

BEGIN
  errCode := MPPOpen;
  IF errCode <> noErr THEN
    WRITELN('Error in opening AppleTalk')
    {Maybe serial port B isn't available for use by AppleTalk}
  ELSE
```

```
BEGIN
  {Call Memory Manager to allocate ABusRecord}
  myABRecord := ABRecHandle(NewHandle(nbpSize));
  {Set up our entity name to register}
  WITH myEntity DO
    BEGIN
      objStr := 'Gene Station'; {we are called 'Gene Station' }
      typeStr := 'PrintSpooler'; { and are of type 'PrintSpooler'}
      zoneStr := '*';
      {Allocate data space for the entity name (used by NBP)}
      nbpNamePtr := NewPtr(LENGTH(objStr) + LENGTH(typeStr) +
                           LENGTH(zoneStr) + 12);
    END;
  {Set up the ABusRecord for the NBPRegister call}
  WITH myABRecord^^ DO
    BEGIN
      nbpEntityPtr := @myEntity;
      nbpBufPtr := nbpNamePtr; {buffer used by NBP internally}
      nbpBufSize := nbpNameBufSize;
      nbpAddress.aSocket := mySocket; {socket to register us on}
      nbpRetransmitInfo.retransInterval := 8; {retransmit every 64 }
      nbpRetransmitInfo.retransCount := 3; { ticks and try 3 times}
    END;
  async := FALSE;
  errCode := NBPRegister(myABRecord, async);
  IF errCode <> noErr THEN
    WRITELN('Error occurred in the NBPRegister call')
    {Maybe the name is already registered somewhere else on the }
    { network.}
  ELSE
    BEGIN
      {Now that we've registered our name, find others of type }
      { 'PrintSpooler'.}
      WITH myEntity DO
        BEGIN
          objStr := '='; {any one of type }
          typeStr := 'PrintSpooler'; { "PrintSpooler" }
          zoneStr := '*'; { in our zone}
        END;
      WITH myABRecord^^ DO
        BEGIN
          nbpEntityPtr := @myEntity;
          nbpBufPtr := @myBuffer; {buffer to place responses in}
          nbpBufSize := SIZEOF(myBuffer);
         {The field nbpDataField, before the NBPLookup call, represents an }
          { approximate number of responses. After the call, nbpDataField }
          { contains the actual number of responses received.}
          nbpDataField := 100; {we want about 100 responses back}
        END;
      errCode := NBPLookup(myABRecord, async); {make sync call}
      IF errCode <> noErr THEN
        WRITELN('An error occurred in the NBPLookup')
        {Did the buffer overflow?}
      ELSE
        BEGIN
```

```
            {Get the first reply}
            errCode := NBPExtract(@mybuffer, myABRecord^^.nbpDataField, 1,
                                  myEntity, entityAddr);
          {The socket address and name of the entity are returned here. If we }
            { want all of them, we'll have to loop for each one in the buffer.}
            IF errCode <> noErr THEN WRITELN('Error in NBPExtract');
            {Maybe the one we wanted wasn't in the buffer}
          END;
        END;
    END;
END.
```

Miscellaneous Routines

FUNCTION GetNodeAddress (VAR myNode,myNet: INTEGER) : OSErr; [Not in ROM]

GetNodeAddress returns the current node ID and network number of the caller. If the .MPP driver isn't installed, it returns noMPPErr. If myNet contains 0, this means that a bridge hasn't yet been found.

Result codes		
	noErr	No error
	noMPPErr	MPP driver not installed

FUNCTION IsMPPOpen : BOOLEAN; [Not in ROM]

IsMPPOpen returns TRUE if the .MPP driver is loaded and running.

FUNCTION IsATPOpen : BOOLEAN; [Not in ROM]

IsATPOpen returns TRUE if the .ATP driver is loaded and running.

NEW APPLETALK MANAGER PASCAL INTERFACE

In addition to the interface documented in the previous section, a new parameter block–style interface to the AppleTalk Manager is now available for Pascal programmers. This new interface, referred to as the preferred interface, is available in addition to the Pascal interface described in the previous section, which is referred to as the alternate interface. All AppleTalk Manager calls, old and new, are supported by the preferred interface.

The alternate interface has not been extended to support the new AppleTalk Manager calls. However, the alternate interface provides the only implementation of LAPRead and DDPRead. These are higher-level calls not directly supported through the assembly-language interface. Developers will wish to use the alternate interface for these calls, and also for compatibility with previous applications. In all other cases, it is recommended that the new preferred interface be used.

Using Pascal

All AppleTalk Manager calls in the preferred interface are essentially equivalent to the corresponding assembly-language calls. Their form is

```
FUNCTION MPPCall (pbPtr: Ptr; asyncFlag: BOOLEAN) : OSErr;
```

where pbPtr points to a device manager parameter block, and asyncFlag is TRUE if the call is to be executed asynchronously. Three parameter block types are provided by the preferred interface (MPP, ATP, and XPP). The MPP parameter block is shown below. The ATP parameter block is shown in the following section, and the XPP parameter block is shown in the "Calling the .XPP Driver" section of this document. The field names in these parameter blocks are the same as the parameter block offset names defined in the assembly-language section (except as documented below). The caller fills in the parameter block with the fields as specified in that section and issues the appropriate call. The interface issues the actual device manager control call.

On asynchronous calls, the caller may pass a completion routine pointer in the parameter block, at offset ioCompletion. This routine will be executed upon completion of the call. It is executed at interrupt level and must not make any memory manager calls. If it uses application globals, it must ensure that register A5 is set up correctly; for details see SetupA5 and RestoreA5 in the Operating System Utilities chapter. If no completion routine is desired, ioCompletion should be set to NIL.

Asynchronous calls return control to the caller with result code of noErr as soon as they are queued to the driver. This isn't an indication of successful completion. To determine when the call is actually completed, if you don't want to use a completion routine, you can poll the ioResult field; this field is set to 1 when the call is made, and receives the actual result code upon completion.

Refer to the appropriate sections of this chapter for the parameter blocks used by each MPP and ATP call. As different MPP and ATP calls take different arguments in their parameter block, two Pascal variant records have been defined to account for all the different cases. These parameter blocks are shown in the sections that follow. The first four fields (which are the same for all calls) are automatically filled in by the device manager. The csCode and ioRefnum fields are automatically filled in by the interface, depending on which call is being made, except in XPP where the caller must fill in the ioRefnum. The ioVRefnum field is unused.

There are two fields that at the assembly-language level have more than one name. These two fields have been given only one name in the preferred interface. These are entityPtr and ntqelPtr, which are both referred to as entityPtr, and atpSocket and currBitmap, which are both referred to as atpSocket. These are the only exceptions to the naming convention.

MPP Parameter Block

```
MPPParamBlock = PACKED RECORD
       qLink:          QElemPtr;     {next queue entry}
       qType:          INTEGER;      {queue type}
       ioTrap:         INTEGER;      {routine trap}
       ioCmdAddr:      Ptr;          {routine address}
       ioCompletion:   ProcPtr;      {completion routine}
       ioResult:       OSErr;        {result code}
       ioNamePtr:      StringPtr;    {command result (ATP user bytes) [long]}
       ioVRefNum:      INTEGER;      {volume reference or drive number}
       ioRefNum:       INTEGER;      {driver reference number}
       csCode:         INTEGER;      {call command code AUTOMATICALLY SET}

       CASE MPPParmType OF
       LAPWriteParm:
                     (filler0:INTEGER;
                     wdsPointer:Ptr);    {->Write Data Structure}
       AttachPHParm,DetachPHParm:
                     (protType:Byte;     {ALAP Protocol Type}
                     filler1:Byte;
                     handler:Ptr);       {->protocol handler routine}
       OpenSktParm,CloseSktParm,WriteDDPParm:
                     (socket:Byte;       {socket number}
                     checksumFlag:Byte;  {checksum flag}
                     listener:Ptr);      {->socket listener routine}
       RegisterNameParm,LookupNameParm,ConfirmNameParm,RemoveNameParm:
                     (interval:Byte;     {retry interval}
                     count:Byte;         {retry count}
                     entityPtr:Ptr;      {->names table element or }
                                         { ->entity name}
                     CASE MPPParmType OF
                     RegisterNameParm:
                              (verifyFlag:Byte;     {set if verify needed}
                               filler3:Byte);
                     LookupNameParm:
                                    (retBuffPtr:Ptr;       {->return buffer}
                                    retBuffSize:INTEGER;   {return buffer size}
                                    maxToGet:INTEGER;      {matches to get}
```

```
                                numGotten:INTEGER);    {matched gotten}
                    ConfirmNameParm:
                                (confirmAddr:AddrBlock; {->entity}
                                newSocket:Byte;         {socket number}
                                filler4:Byte));

        SetSelfSendParm:
                    (newSelfFlag:Byte;  {self-send toggle flag}
                    oldSelfFlag:Byte);  {previous self-send state}
        KillNBPParm:
                    (nKillQEl:Ptr);     {ptr to Q element to cancel}
      END;

ATP Parameter Block

ATPParamBlock = PACKED RECORD
          qLink:            QElemPtr;    {next queue entry}
          qType:            INTEGER;     {queue type}
          ioTrap:           INTEGER;     {routine trap}
          ioCmdAddr:        Ptr;         {routine address}
          ioCompletion:     ProcPtr;     {completion routine}
          ioResult:         OSErr;       {result code}
          userData:         LONGINT;     {ATP user bytes [long]}
          reqTID:           INTEGER;     {request transaction ID}
          ioRefNum:         INTEGER;     {driver reference number
          csCode:           INTEGER;     {Call command code }
                                         { AUTOMATICALLY SET}
          atpSocket:        Byte;        {currBitMap or socket number}
          atpFlags:         Byte;        {control information}
          addrBlock:        AddrBlock;   {source/dest. socket address}
          reqLength:        INTEGER;     {request/response length}
          reqPointer:       Ptr;         {-> request/response data}
          bdsPointer:       Ptr;         {-> response BDS}
          CASE MPPParmType OF
                SendRequestParm,NSendRequestParm:
                    (numOfBuffs:Byte;   {numOfBuffs}
                    timeOutVal:Byte;    {timeout interval}
                    numOfResps:Byte;    {number responses actually received}
                    retryCount:Byte;    {number of retries}
                    intBuff:INTEGER);   {used internally for NSendRequest}
                SendResponseParm:
                    (filler0:Byte;      {number of responses being sent}
                    bdsSize:Byte;       {number of BDS elements}
                    transID:INTEGER);   {transaction ID}
                GetRequestParm:
                    (bitMap:Byte;       {bit map}
                    filler1:Byte);
                AddResponseParm:
                    (rspNum:Byte;       {sequence number}
                    filler2:Byte);
                KillSendReqParm,KillGetReqParm:
                    (aKillQEl:Ptr);     {ptr to Q element to cancel}
          END;
```

The following table is a complete list of all the parameter block calls provided by the preferred interface.

AppleTalk Manager Routine	Preferred Interface Call
AttachPH	`Function PAttachPH (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;`
DetachPH	`Function PDetachPH (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;`
WriteLAP	`Function PWriteLAP (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;`
OpenSkt	`Function POpenSkt (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;`
CloseSkt	`Function PCloseSkt (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;`
WriteDDP	`Function PWriteDDP (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;`
RegisterName	`Function PRegisterName (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;`
LookupName	`Function PLookupName (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;`
ConfirmName	`Function PConfirmName (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;`
RemoveName	`Function PRemoveName (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;`
OpenATPSkt	`Function POpenATPSkt (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;`
CloseATPSkt	`Function PCloseATPSkt (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;`
SendRequest	`Function PSendRequest (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;`
GetRequest	`Function PGetRequest (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;`
SendResponse	`Function PSendResponse (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;`
AddResponse	`Function PAddResponse(thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;`
ReltCB	`Function PRelTCB (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;`
RelRspCB	`Function PRelRspCB (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;`
SetSelfSend	`Function PSetSelfSend (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;`
NSendRequest	`Function PNSendRequest (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;`
KillSendReq	`Function PKillSendReq (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;`
KillGetReq	`Function PKillGetReq (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;`
KillNBP	`Function PKillNBP (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;`

Building Data Structures

Because it is difficult for Pascal to deal with certain assembly-language structures, the preferred interface provides a number of routines for building these structures. These routines are summarized below.

```
PROCEDURE BuildLAPwds (wdsPtr,dataPtr: Ptr;
                       destHost,protoType,frameLen: INTEGER);
```

This routine builds a single-frame write data structure LAP WDS for use with the PWriteLAP call. Given a buffer of length frameLen pointed to by dataPtr, it fills in the WDS pointed to by wdsPtr and sets the destination node and protocol type as indicated by destHost and protoType, respectively. The WDS indicated must contain at least two elements.

```
PROCEDURE BuildDDPwds  (wdsPtr,headerPtr,dataPtr: Ptr; destAddress: AddrBlock;
                        DDPType : INTEGER; dataLen: INTEGER);
```

This routine builds a single-frame write data structure DDP WDS, for use with the PWriteDDP call. Given a header buffer of at least 17 bytes pointed to by headerPtr and a data buffer of length dataLen pointed to by dataPtr, it fills in the WDS pointed to by wdsPtr, and sets the destination address and protocol type as indicated by destaddress and DDPtype, respectively. The WDS indicated must contain at least 3 elements.

```
PROCEDURE NBPSetEntity (buffer: Ptr; nbpObject,nbpType,nbpZone: Str32);
```

This routine builds an NBP entity structure, for use with the PLookupNBP and PConfirmName calls. Given a buffer of at least the size of the EntityName data structure (99 bytes) pointed to by buffer, this routine sets the indicated object, type, and zone in that buffer.

```
PROCEDURE NBPSetNTE (ntePtr: Ptr; nbpObject,nbpType,nbpZone: Str32;
                     Socket: INTEGER);
```

This routine builds an NBP names table entry, for use with the PRegisterName call. Given a names table entry of at least the size of the EntityName data structure plus nine bytes (108 bytes) pointed to by ntePtr, this routine sets the indicated object, type, zone, and socket in that names table entry.

```
FUNCTION NBPExtract (theBuffer: Ptr; numInBuf: INTEGER; whichOne: INTEGER; VAR
abEntity: EntityName; VAR address: AddrBlock) : OSErr;
```

This routine is provided in the alternate interface, but can be used as provided for extracting NBP entity names from a look-up response buffer.

```
FUNCTION GetBridgeAddress: INTEGER;
```

This routine returns the current address of a bridge in the low byte, or zero if there is none.

```
FUNCTION BuildBDS (buffPtr,bdsPtr: Ptr; buffSize: INTEGER) : INTEGER;
```

This routine builds a BDS, for use with the ATP calls. Given a data buffer of length buffSize pointed to by buffPtr, it fills in the BDS pointed to by bdsPtr. The buffer will be broken up into pieces of maximum size (578 bytes). The user bytes in the BDS are not modified by this routine. This routine is provided only as a convenience; generally the caller will be able to build the BDS completely from Pascal without it.

PICKING A NODE ADDRESS IN THE SERVER RANGE

Normally upon opening, the node number picked by the AppleTalk manager will be in the node number range ($01–$7F). It is possible to indicate that a node number in the server range ($80–$FE) is desired. Picking a number in the server range is a more time-consuming but more thorough process, and it's required for server nodes because it greatly decreases the possibility of a node number conflict.

To open AppleTalk with a server node number, an extended open call is used. An extended open call is indicated by having the immediate bit set in the Open trap itself. In the extended open call, the high bit (bit 31) of the extension longword field (ioMix) indicates whether a server or workstation node number should be picked. Set this bit to 1 to request a server node number. The rest of this field should be zero, as should all other unused fields in the queue element. A server node number can only be requested on the first Open call to the .MPP driver.

SENDING PACKETS TO ONE'S OWN NODE

Upon opening, the ability to send a packet to one's own node (intranode delivery) is disabled. This feature of the AppleTalk Manager can be manipulated through the SetSelfSend function. Once enabled, it is possible, at all levels, to send packets to entities within one's own node. An example of where this might be desirable is an application sending data to a print spooler that is actually running in the background on the same node.

Enabling (or disabling) this feature affects the entire node and should be performed with care. For instance, a desk accessory may not expect to receive names from within its own node as a response to an NBP look-up; enabling this feature from an application could break the desk accessory. All future programs should be written with this feature in mind.

```
FUNCTION PSetSelfSend (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;
```

Parameter Block

-->	26	csCode	word	Always PSetSelfSend
-->	28	newSelfFlag	byte	New SelfSend flag
<--	29	oldSelfFlag	byte	Old SelfSend flag

PSetSelfSend enables or disables the intranode delivery feature of the AppleTalk Manager. If newSelfFlag is nonzero, the feature will be enabled; otherwise it will be disabled. The previous value of the flag will be returned in oldSelfFlag.

Result Codes	noErr	No error

ATP DRIVER CHANGES

Changes to the ATP driver include the ability to send an ATP request through a specific socket rather than having ATP open a new socket, a new call to abort outstanding SendRequest calls, and a new call to abort specific outstanding GetRequest calls.

Sending an ATP Request Through a Specified Socket

ATP requests can now be sent through client-specified sockets. ATP previously would open a dynamic socket, send the request through it, and close the socket when the request was completed. The client can now choose to send a request through an already-opened socket; this also allows more than one request to be sent per socket. A new call, PNSendRequest, has been added for this purpose. The function of the old SendRequest call itself remains unchanged.

```
FUNCTION PNSendRequest (thePBptr: ATPBPtr; async: BOOLEAN) : OSErr;
```

Parameter block

-->	18	userData	longword	User bytes
<--	22	reqTID	word	Transaction ID used in request
-->	26	csCode	word	Always sendRequest
<->	28	atpSocket	byte	Socket to send request on or current bitmap
<->	29	atpFlags	byte	Control information
-->	30	addrBlock	longword	Destination socket address
-->	34	reqLength	word	Request size in bytes
-->	36	reqPointer	pointer	Pointer to request data
-->	40	bdsPointer	pointer	Pointer to response BDS
-->	44	numOfBuffs	byte	Number of responses expected
-->	45	timeOutVal	byte	Timeout interval
<--	46	numOf Resps	byte	Number of responses received
<->	47	retryCount	byte	Number of retries
<--	48	intBuff	word	Used internally

The PNSendRequest call is functionally equivalent to the SendRequest call, however PNSendRequest allows you to specify, in the atpSocket field, the socket through which the request is to be sent. This socket must have been previously opened through an OpenATPSkt request (otherwise a badATPSkt error will be returned). Note that PNSendRequest requires two additional bytes of memory at the end of the parameter block, immediately following the retryCount. These bytes are for the internal use of the AppleTalk Manager and should not be modified while the PNSendRequest call is active.

There is a machine-dependent limit as to the number of concurrent PNSendRequests that can be active on a given socket. If this limit is exceeded, the error tooManyReqs is returned.

One additional difference between SendRequest and PNSendRequest is that a PNSendRequest can only be aborted by a PKillSendReq call (see below), whereas a SendRequest can be aborted by either a RelTCB or KillSendReq call.

Result Codes	noErr	No error
	reqFailed	Retry count exceeded
	tooManyReqs	Too many concurrent requests
	noDataArea	Too many outstanding ATP calls
	reqAborted	Request cancelled by user

Aborting ATP SendRequests

The RelTCB call is still supported, but only for aborting SendRequests. To abort PNSendRequests, a new call, PKillSendReq, has been added. This call will abort both SendRequests and PNSendRequests. PKillSendReq's only argument is the queue element pointer of the request to be aborted. The queue element pointer is passed at the offset of the PKillSendReq queue element specified by aKillQE1.

```
FUNCTION PKillSendReq (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;
```

Parameter block

-->	26	csCode	word	Always PKillSendReq
-->	44	aKillQEl	pointer	Pointer to queue element

PKillSendReq is functionally equivalent to RelTCB, except that it takes different arguments and will abort both SendRequests and PNSendRequests. To abort one of these calls, place a pointer to the queue element of the call to abort in aKillQEl and issue the PKillSendReq call.

Result Codes	noErr	No error
	cbNotFound	aKillQEl does not point to a SendReq or NSendReq queue element

---

Aborting ATP GetRequests

ATP GetRequests can now be aborted through the PKillGetReq call. This call looks and works just like the PKillSendReq call, and is used to abort a specific GetRequest call. Previously it was necessary to close the socket to abort all GetRequest calls on the socket.

```
FUNCTION PKillGetReq (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;
```

Parameter block

-->	26	csCode	word	Always PKillGetReq
-->	44	aKillQEl	pointer	Pointer to queue element

PKillGetReq will abort a specific outstanding GetRequest call (as opposed to closing the socket, which aborts all outstanding GetRequests on that socket). The call will be completed with a reqAborted error. To abort a GetRequest, place a pointer to the queue element of the call to abort in aKillQEl and issue the PKillGetReq call.

Result Codes	noErr	No error
	cbNotFound	aKillQEl does not point to a GetReq queue element

NAME BINDING PROTOCOL CHANGES

Changes to the Name Binding Protocol include supporting multiple concurrent requests and a means for aborting an active request.

Multiple Concurrent NBP Requests

NBP now supports multiple concurrent active requests. Specifically, a number of LookupNames, RegisterNames and ConfirmNames can all be active concurrently. The maximum number of concurrent requests is machine dependent; if it is exceeded the error tooManyReqs will be returned. Active requests can be aborted by the PKillNBP call.

KillNBP function

```
FUNCTION PKillNBP (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;
```

•••Click on the X-Ref button, and refer to Technical Note #199.•••

Parameter block

-->	26	csCode	word	Always PKillNBP
-->	28	aKillQEl	pointer	Pointer to queue element

PKillNBP is used to abort an outstanding LookupName, RegisterName or ConfirmName request. To abort one of these calls, place a pointer to the queue element of the call to abort in a KillQEl and issue the PKillNBP call. The call will be completed with a ReqAborted error.

Result Codes	noErr	No error
	cbNotFound	aKillQEl does not point to a valid NBP queue element

VARIABLE RESOURCES

The table below lists machine-dependent resources for the different Macintosh system configurations. The RAM-based resources are available through the AppleShare Server.

Resource	Macintosh Plus	RAM-Based	Macintosh SE	Macintosh II
Protocol Handlers	4	8	8	8
Statically Assigned Sockets	14*	12	12	14
Concurrent ATP SendRequests	6	12	12	12
ATP Sockets	6	32	32	126
Concurrent ATP Responses	8	16	16	32
Concurrent NBP Requests	1	6	6	10
Concurrent ASP Sessions	N/A	5	10	20
Concurrent ATP NSendRequests Per Socket **	N/A	9	14	62

* Includes dynamic sockets
** Determined dynamically at runtime based on CPU speed.
N/A : Not Applicable

CALLING THE APPLETALK MANAGER FROM ASSEMBLY LANGUAGE

This section discusses how to use the AppleTalk Manager from assembly language. Equivalent Pascal information is given in the preceding section.

All routines make Device Manager Control calls. The description of each routine includes a list of the fields needed. Some of these fields are part of the parameter block described in the Device Manager chapter; additional fields are provided for the AppleTalk Manager.

The number next to each field name indicates the byte offset of the field from the start of the parameter block pointed to by A0. An arrow next to each parameter name indicates whether it's an input, output, or input/output parameter:

Arrow	Meaning
-->	Parameter is passed to the routine
<--	Parameter is returned by the routine
<->	Parameter is passed to and returned by the routine

All Device Manager Control calls return an integer result code of type OSErr in the ioResult field. Each routine description lists all of the applicable result codes generated by the AppleTalk Manager, along with a short description of what the result code means. Lengthier explanations of all the result codes can be found in the summary at the end of this chapter. Result codes from other parts of the Operating System may also be returned. (See Appendix A for a list of all result codes.)

Opening AppleTalk

•••Click on the X-Ref button, and refer to Technical Note #224.•••

Two tests are made at system startup to determine whether the .MPP driver should be opened at that time. If port B is already in use, or isn't configured for AppleTalk, .MPP isn't opened until explicitly requested by an application; otherwise it's opened at system startup.

It's the application's responsibility to test the availability of port B before opening AppleTalk. Assembly-language programmers can use the Pascal calls MPPOpen and ATPLoad to open the .MPP and .ATP drivers.

The global variable SPConfig is used for configuring the serial ports; it's copied from a byte in parameter RAM (which is discussed in the Operating System Utilities chapter). The low-order four bits of this variable contain the current configuration of port B. The following use types are provided as global constants for testing or setting the configuration of port B:

```
useFree     .EQU    0    ;unconfigured
useATalk    .EQU    1    ;configured for AppleTalk
```

```
  useAsync    .EQU    2    ;configured for the Serial Driver
```

The application shouldn't attempt to open AppleTalk unless SPConfig is equal to either useFree or useATalk.

A second test involves the global variable PortBUse; the low-order four bits of this byte are used to monitor the current use of port B. If PortBUse is negative, the program is free to open AppleTalk. If PortBUse is positive, the program should test to see whether port B is already being used by AppleTalk; if it is, the low-order four bits of PortBUse will be equal to the use type useATalk.

The .MPP driver sets PortBUse to the correct value (useATalk) when it's opened and resets it to $FF when it's closed. Bits 4-6 of this byte are used for driver-specific information; ATP uses bit 4 to indicate whether it's currently opened:

```
atpLoadedBit    .EQU    4    ;set if ATP is opened
```

Example

The following code illustrates the use of the SPConfig and PortBUse variables.

```
          MOVE      #-<atpUnitNum+1>,atpRefNum(A0)  ;save known ATP refNum in
                                                    ; case ATP not opened
OpenAbus  SUB       #ioQElSize,SP     ;allocate queue entry
          MOVE.L    SP,A0             ;A0 -> queue entry
          CLR.B     ioPermssn(A0)     ;make sure permission's clear
          MOVE.B    PortBUse,D1       ;is port B in use?
          BPL.S     @10               ;if so, make sure by AppleTalk
          MOVEQ     #portNotCf,D0     ;assume port not configured for AppleTalk
          MOVE.B    SPConfig,D1       ;get configuration data
          AND.B     #$0F,D1           ;mask it to low 4 bits
          SUBQ.B    #useATalk,D1      ;unconfigured or configured for AppleTalk
          BGT.S     @30               ;if not, return error
          LEA       mppName,A1        ;A1 = address of driver name
          MOVE.L    A1,ioFileName(A0) ;set in queue entry
          _Open                       ;open MPP
          BNE.S     @30               ;return error, if it can't load it
          BRA.S     @20               ;otherwise, go check ATP
@10       MOVEQ     #portInUse,D0     ;assume port in use error
          AND.B     #$0F,D1           ;clear all but use bits
          SUBQ.B    #useATalk,D1      ;is AppleTalk using it?
          BNE.S     @30               ;if not, then error
@20       MOVEQ     #0,D0             ;assume no error
          BTST      #atpLoadedBit,PortBUse    ;ATP already open?
          BNE.S     @30               ;just return if so
          LEA       atpName,A1        ;A1 = address of driver name
          MOVE.L    A1,ioFileName(A0) ;set in queue entry
          _Open                       ;open ATP
@30       ADD       #ioQElSize,SP     ;deallocate queue entry
          RTS                         ;and return
mppName   .BYTE     4                 ;length of .MPP driver name
          .ASCII    '.MPP'            ;name of .MPP driver
atpName   .BYTE     4                 ;length of .ATP driver name
```

```
          .ASCII    '.ATP'            ;name of .ATP driver
```

Data Structures

An ALAP frame is composed of a three-byte header, up to 600 bytes of data, and a two-byte frame check sequence (Figure 6). You can use the following global constants to access the contents of an ALAP header:

```
  lapDstAdr  .EQU    0    ;destination node ID
  lapSrcAdr  .EQU    1    ;source node ID
  lapType    .EQU    2    ;ALAP protocol type
  lapHdSz    .EQU    3    ;ALAP header size
```

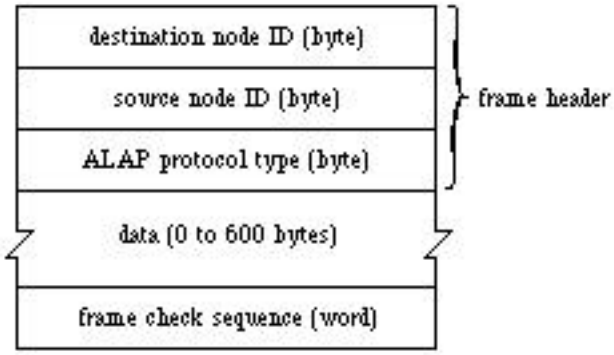

Figure 6–ALAP Frame

Figure 6–ALAP Frame

Two of the protocol handlers in every node are used by DDP. These protocol handlers service frames with ALAP protocol types equal to the following global constants:

```
  shortDDP   .EQU    1    ;short DDP header
  longDDP    .EQU    2    ;long DDP header
```

When you call ALAP to send a frame, you pass it information about the frame in a write data structure, which has the format shown in Figure 7.

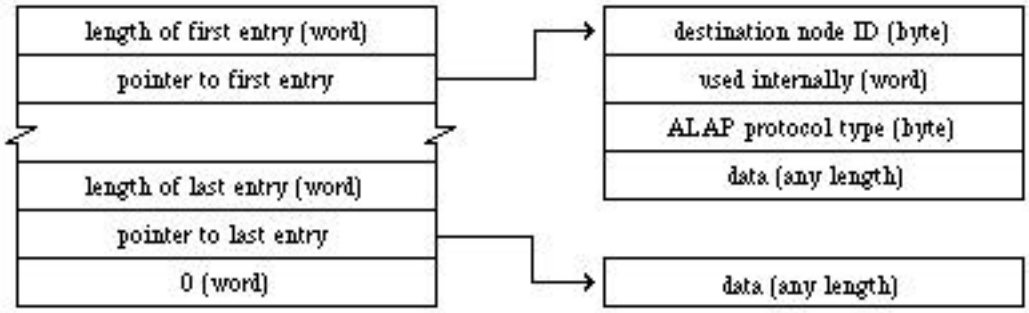

Figure 7–Write Data Structure for ALAP

Figure 7–Write Data Structure for ALAP

If you specify a destination node ID of 255, the frame will be broadcast to all nodes. The byte that's "used internally" is used by the AppleTalk Manager to store the address of the node sending the frame.

Using ALAP

Most programs will never need to call ALAP, because higher-level protocols will automatically call ALAP as necessary. If you do want to send a frame directly via an ALAP, call the WriteLAP function. There's no ReadLAP function in assembly language; if you want to read ALAP frames, you must call AttachPH to add your protocol handler to the node's protocol handler table. The ALAP module will examine every incoming frame and call your protocol handler for each frame received with the correct ALAP protocol. When your program no longer wants to receive frames with a particular ALAP protocol type value, it can call DetachPH to remove the corresponding protocol handler from the protocol handler table.

See the "Protocol Handlers and Socket Listeners" section for information on how to write a protocol handler.

ALAP Routines

WriteLAP function

```
Parameter block
  -->    26    csCode      word     ;always writeLAP
  -->    30    wdsPointer  pointer  ;write data structure
```

WriteLAP sends a frame to another node. The frame data and destination of the frame are described by the write data structure pointed to by wdsPointer. The first two data bytes of an ALAP frame sent to another computer using the AppleTalk Manager must indicate the length of the frame in bytes. The ALAP protocol type byte must be in the range 1 to 127.

```
Result codes    noErr           No error
                excessCollsns   No CTS received after 32 RTS's
                ddpLengthErr    Packet length exceeds maximum
                lapProtErr      Invalid ALAP protocol type
```

AttachPH function

```
Parameter block
  -->    26    csCode    word     ;always attachPH
  -->    28    protType  byte     ;ALAP protocol type
  -->    30    handler   pointer  ;protocol handler
```

AttachPH adds the protocol handler pointed to by handler to the node's protocol table. ProtType specifies what kind of frame the protocol handler can service. After AttachPH is called, the protocol handler is called for each incoming frame whose ALAP protocol type equals protType.

```
Result codes    noErr         No error
                lapProtErr    Error attaching protocol type
```

DetachPH function

```
Parameter block
  -->    26    csCode    word  ;always detachPH
  -->    28    protType  byte  ;ALAP protocol type
```

DetachPH removes from the node's protocol table the specified ALAP protocol type and corresponding protocol handler.

```
Result codes    noErr         No error
                lapProtErr    Error detaching protocol type
```

Datagram Delivery Protocol

Data Structures

A DDP datagram consists of a header followed by up to 586 bytes of actual data (Figure 8). The headers can be of two different lengths; they're identified by the following ALAP protocol types:

```
  shortDDP   .EQU    1    ;short DDP header
  longDDP    .EQU    2    ;long DDP header
```

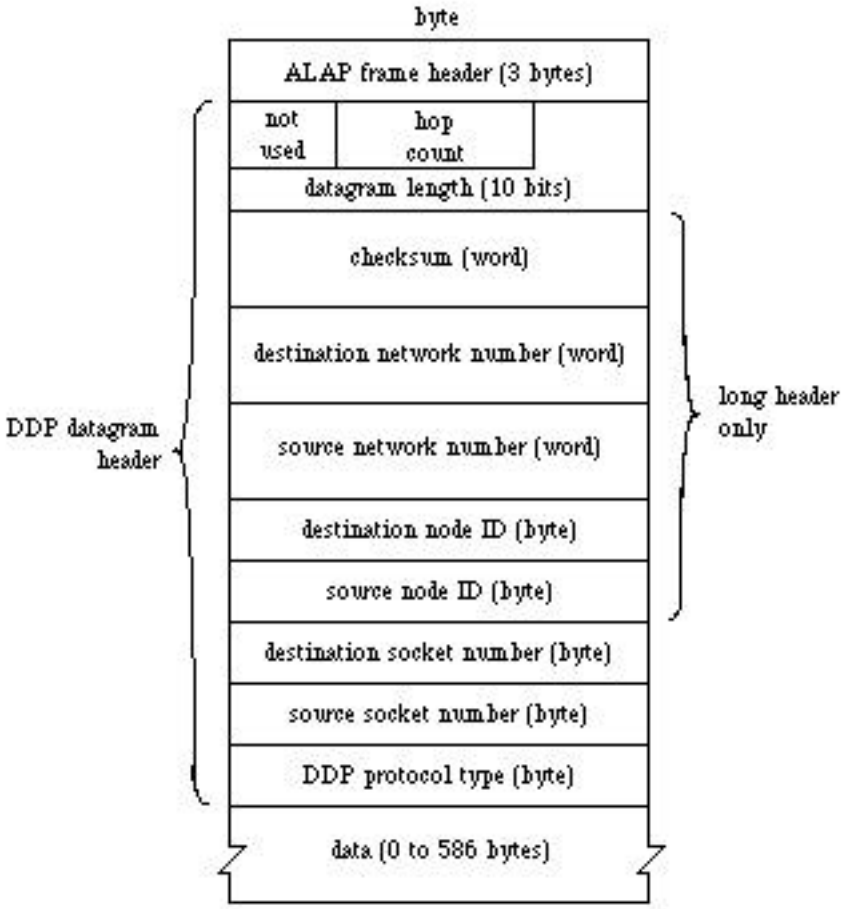

Figure 8–DDP Datagram

Figure 8–DDP Datagram

Long DDP headers (13 bytes) are used for sending datagrams between two or more different AppleTalk networks. You can use the following global constants to access the contents of a long DDP header:

```
ddpHopCnt    .EQU    0    ;count of bridges passed (4 bits)
ddpLength    .EQU    0    ;datagram length (10 bits)
ddpChecksum  .EQU    2    ;checksum
ddpDstNet    .EQU    4    ;destination network number
ddpSrcNet    .EQU    6    ;source network number
ddpDstNode   .EQU    8    ;destination node ID
ddpSrcNode   .EQU    9    ;source node ID
ddpDstSkt    .EQU    10   ;destination socket number
ddpSrcSkt    .EQU    11   ;source socket number
ddpType      .EQU    12   ;DDP protocol type
```

The size of a DDP long header is given by the following constant:

```
ddpHSzLong   .EQU    ddpType+1
```

The short headers (five bytes) are used for datagrams sent to sockets within the same network as the source socket. You can use the following global constants to access the contents of a short DDP header:

```
ddpLength    .EQU    0              ;datagram length
sDDPDstSkt   .EQU    ddpChecksum    ;destination socket number
sDDPSrcSkt   .EQU    sDDPDstSkt+1   ;source socket number
sDDPType     .EQU    sDDPSrcSkt+1   ;DDP protocol type
```

The size of a DDP short header is given by the following constant:

```
ddpHSzShort  .EQU    sDDPType+1
```

The datagram length is a ten-bit field. You can use the following global constant as a mask for these bits:

```
ddpLenMask   .EQU    $03FF
```

The following constant indicates the maximum length of a DDP datagram:

```
ddpMaxData   .EQU    586
```

When you call DDP to send a datagram, you pass it information about the datagram in a write data structure with the format shown in Figure 9.

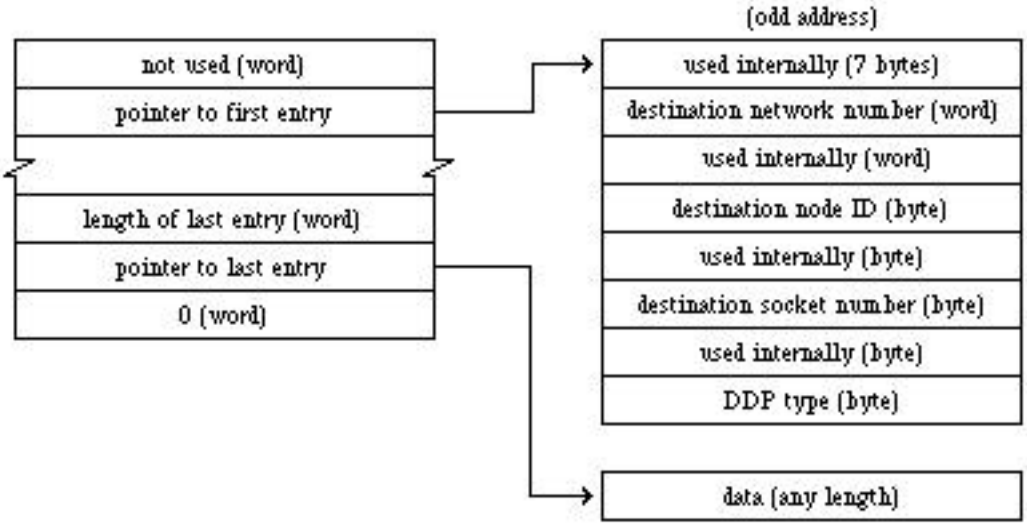

Figure 9–Write Data Structure for DDP

Figure 9–Write Data Structure for DDP

The first seven bytes are used internally for the ALAP header and the DDP datagram length and checksum. The other bytes used internally store the network number, node ID, and socket number of the socket client sending the datagram.

Warning: The first entry in a DDP write data structure must begin at an odd address.

If you specify a node ID of 255, the datagram will be broadcast to all nodes within the destination network. A network number of 0 means the local network to which the node is connected.

Warning: DDP always destroys the high-order byte of the destination network number when it sends a datagram with a short header. Therefore, if you want to reuse the first entry of a DDP write data structure entry, you must restore the destination network number.

Using DDP

Before it can use a socket, the program must call OpenSkt, which adds a socket and its socket listener to the socket table. When a client is finished using a socket, call CloseSkt, which removes the socket's entry from the socket table. To send a datagram via DDP, call WriteDDP. If you want to read DDP datagrams, you must write your own socket listener. DDP will send every incoming datagram for that socket to your socket listener.

See the "Protocol Handlers and Socket Listeners" section for information on how to write a socket listener.

DDP Routines

OpenSkt function

```
Parameter block
  -->    26    csCode    word     ;always openSkt
  <->    28    socket    byte     ;socket number
  -->    30    listener  pointer  ;socket listener
```

OpenSkt adds a socket and its socket listener to the socket table. If the socket parameter is nonzero, it must be in the range 64 to 127, and it specifies the socket's number; if socket is 0, OpenSkt opens a socket with a socket number in the range 128 to 254, and returns it in the socket parameter. Listener contains a pointer to the socket listener.

OpenSkt will return ddpSktErr if you pass the number of an already opened socket, if you pass a socket number greater than 127, or if the socket table is full (the socket table can hold a maximum of 12 sockets).

```
Result codes    noErr        No error
                ddpSktErr    Socket error
```

CloseSkt function

```
Parameter block
  -->    26    csCode  word  ;always closeSkt
  -->    28    socket  byte  ;socket number
```

CloseSkt removes the entry of the specified socket from the socket table. If you pass a socket number of 0, or if you attempt to close a socket that isn't open, CloseSkt will return ddpSktErr.

```
Result codes    noErr        No error
                ddpSktErr    Socket error
```

WriteDDP function

Parameter block

```
  -->    26    csCode        word     ;always writeDDP
  -->    28    socket        byte     ;socket number
  -->    29    checksumFlag  byte     ;checksum flag
  -->    30    wdsPointer    pointer  ;write data structure
```

WriteDDP sends a datagram to another socket. WDSPointer points to a write data structure containing the datagram and the address of the destination socket. If checksumFlag is TRUE, WriteDDP will compute the checksum for all datagrams requiring long headers.

Result codes		
	noErr	No error
	ddpLenErr	Datagram length too big
	ddpSktErr	Socket error
	noBridgeErr	No bridge found

AppleTalk Transaction Protocol

Data Structures

An ATP packet consists of an ALAP header, DDP header, and ATP header, followed by actual data (Figure 10). You can use the following global constants to access the contents of an ATP header:

```
  atpControl   .EQU    0    ;control information
  atpBitMap    .EQU    1    ;bit map
  atpRespNo    .EQU    1    ;sequence number
  atpTransID   .EQU    2    ;transaction ID
  atpUserData  .EQU    4    ;user bytes
```

The size of an ATP header is given by the following constant:

```
  atpHdSz      .EQU    8
```

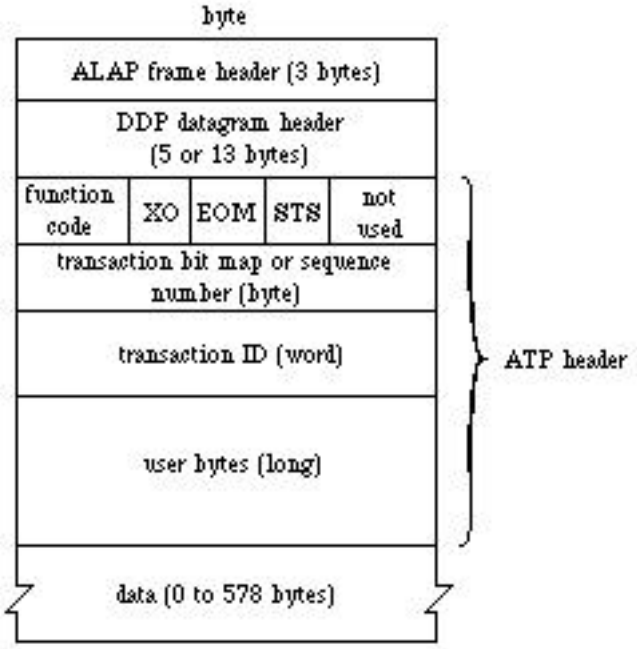

Figure 10–ATP Packet

Figure 10–ATP Packet

ATP packets are identified by the following DDP protocol type:

```
atp          .EQU    3
```

The control information contains a function code and various control bits. The function code identifies either a TReq, TResp, or TRel packet with one of the following global constants:

```
atpReqCode   .EQU    $40    ;TReq packet
atpRspCode   .EQU    $80    ;TResp packet
atpRelCode   .EQU    $C0    ;TRel packet
```

The send-transmission-status, end-of-message, and exactly-once bits in the control information are accessed via the following global constants:

```
atpSTSBit    .EQU    3    ;send-transmission-status bit
atpEOMBit    .EQU    4    ;end-of-message bit
atpXOBit     .EQU    5    ;exactly-once bit
```

Many ATP calls require a field called atpFlags (Figure 11), which contains the above three bits plus the following two bits:

```
sendChk      .EQU    0    ;send-checksum bit
tidValid     .EQU    1    ;transaction ID validity bit
```

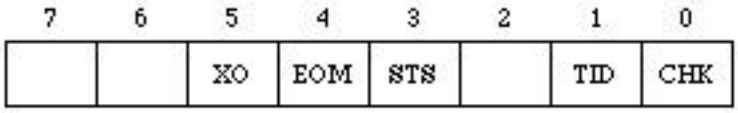

Figure 11–ATPFlags Field

Figure 11–ATPFlags Field

The maximum number of response packets in an ATP transaction is given by the following global constant:

```
atpMaxNum    .EQU    8
```

When you call ATP to send responses, you pass the responses in a response BDS, which is a list of up to eight elements, each of which contains the following:

```
bdsBuffSz    .EQU    0    ;size of data to send
bdsBuffAddr  .EQU    2    ;pointer to data
bdsUserData  .EQU    8    ;user bytes
```

When you call ATP to receive responses, you pass it a response BDS with up to eight elements, each in the following format:

```
bdsBuffSz    .EQU    0    ;buffer size in bytes
bdsBuffAddr  .EQU    2    ;pointer to buffer
bdsDataSz    .EQU    6    ;number of bytes actually received
bdsUserData  .EQU    8    ;user bytes
```

The size of a BDS element is given by the following constant:

```
bdsEntrySz   .EQU    12
```

ATP clients are identified by internet addresses in the form shown in Figure 12.

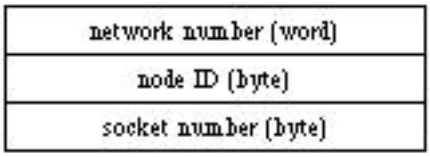

Figure 12–Internet Address

Figure 12–Internet Address

Using ATP

Before you can use ATP on a Macintosh 128K, the .ATP driver must be read from the system resource file via a Device Manager Open call. The name of the .ATP driver is '.ATP' and its reference number is –11. When the .ATP driver is opened, it reads its ATP code into the application heap and installs a task into the vertical retrace queue.

Warning: When another application starts up, the application heap is reinitialized; on a Macintosh 128K, this means that the ATP code is lost (and must be reloaded by the next application).

When you're through using ATP on a Macintosh 128K, call the Device Manager Close routine—the system will be returned to the state it was in before the .ATP driver was opened.

On a Macintosh 512K or XL, the .ATP driver will have been loaded into the system heap either at system startup or upon execution of a Device Manager Open call loading MPP. You shouldn't close the .ATP driver on a Macintosh 512K or XL; AppleTalk expects it to remain open on these systems.

To send a request to another socket and get a response, call SendRequest. The call terminates when either an entire response is received or a specified retry timeout interval elapses. To open a socket for the purpose of responding to requests, call OpenATPSkt. Then call GetRequest to receive a request; when a request is received, the call is completed. After receiving and servicing a request, call SendResponse to return response information. If you cannot or do not want to send the entire response all at once, make a SendResponse call to send some of the response, and then call AddResponse later to send the remainder of the response. To close a socket opened for the purpose of sending responses, call CloseATPSkt.

During exactly-once transactions, SendResponse doesn't terminate until the transaction is completed via a TRel packet, or the retry count is exceeded.

Warning: Don't modify the parameter block passed to an ATP call until the call is completed.

ATP Routines

OpenATPSkt function

```
Parameter block
  -->    26    csCode     word       ;always openATPSkt
  <->    28    atpSocket  byte       ;socket number
  -->    30    addrBlock  long word  ;socket request specification
```

OpenATPSkt opens a socket for the purpose of receiving requests. ATPSocket contains the socket number of the socket to open. If it's 0, a number is dynamically assigned and returned in atpSocket. AddrBlock contains a specification of the socket addresses from which requests will be accepted. A 0 in the network number, node ID, or socket number field of addrBlock means that requests will be accepted from every network, node, or socket, respectively.

```
Result codes    noErr          No error
                tooManySkts    Too many responding sockets
                noDataArea     Too many outstanding ATP calls
```

CloseATPSkt function

```
Parameter block
  -->    26    csCode     word    ;always closeATPSkt
  -->    28    atpSocket  byte    ;socket number
```

CloseATPSkt closes the socket whose number is specified by atpSocket, for the purpose of receiving requests.

```
Result codes    noErr         No error
                noDataArea    Too many outstanding ATP calls
```

SendRequest function

```
Parameter block
  -->    18    userData    long word  ;user bytes
  <--    22    reqTID      word       ;transaction ID used in request
  -->    26    csCode      word       ;always sendRequest
  <--    28    currBitMap  byte       ;bit map
  <->    29    atpFlags    byte       ;control information
  -->    30    addrBlock   long word  ;destination socket address
  -->    34    reqLength   word       ;request size in bytes
  -->    36    reqPointer  pointer    ;pointer to request data
  -->    40    bdsPointer  pointer    ;pointer to response BDS
  -->    44    numOfBuffs  byte       ;number of responses expected
  -->    45    timeOutVal  byte       ;timeout interval
  <--    46    numOfResps  byte       ;number of responses received
  <->    47    retryCount  byte       ;number of retries
```

SendRequest sends a request to another socket and waits for a response. UserData contains the four user bytes. AddrBlock indicates the socket to which the request should be sent. ReqLength and reqPointer contain the size and location of the request to send. BDSPointer points to a response BDS where the responses are to be returned; numOfBuffs indicates the number of responses requested. The number of responses received is returned in numOfResps. If a nonzero value is returned in numOfResps, you can examine currBitMap to determine which packets of the transaction were actually received and to detect pieces for higher-level recovery, if desired.

TimeOutVal indicates the number of seconds that SendRequest should wait for a response before resending the request. RetryCount indicates the maximum number of retries SendRequest should attempt. The end-of-message flag of atpFlags will be set if the EOM bit is set in the last packet received in a valid response sequence. The exactly-once flag should be set if you want the request to be part of an exactly-once transaction.

To cancel a SendRequest call, you need the transaction ID; it's returned in reqTID. You can examine reqTID before the completion of the call, but its contents are valid only after the tidValid bit of atpFlags has been set.

SendRequest completes when either an entire response is received or the retry count is exceeded.

Note: The value provided in retryCount will be modified during SendRequest if any retries are made. This field is used to monitor the number of retries; for each retry, it's decremented by 1.

```
Result codes    noErr          No error
                reqFailed      Retry count exceeded
                tooManyReqs    Too many concurrent requests
```

```
                noDataArea     Too many outstanding ATP calls
                reqAborted     Request canceled by user
```

GetRequest function

```
Parameter block
  <--    18    userData    long word  ;user bytes
  -->    26    csCode      word       ;always getRequest
  -->    28    atpSocket   byte       ;socket number
  <--    29    atpFlags    byte       ;control information
  <--    30    addrBlock   long word  ;source of request
  <->    34    reqLength   word       ;request buffer size
  -->    36    reqPointer  pointer    ;pointer to request buffer
  <--    44    bitMap      byte       ;bit map
  <--    46    transID     word       ;transaction ID
```

GetRequest sets up the mechanism to receive a request sent by a SendRequest call. UserData returns the four user bytes from the request. ATPSocket contains the socket number of the socket that should listen for a request. The internet address of the socket from which the request was sent is returned in addrBlock. ReqLength and reqPointer indicate the size (in bytes) and location of a buffer to store the incoming request. The actual size of the request is returned in reqLength. The transaction bit map and transaction ID will be returned in bitMap and transID. The exactly-once flag in atpFlags will be set if the request is part of an exactly-once transaction.

GetRequest completes when a request is received.

```
Result codes    noErr        No error
                badATPSkt    Bad responding socket
```

SendResponse function

```
Parameter block
  <--    18    userData    long word  ;user bytes from TRel
  <--    22    reqTID      word       ;transaction ID used in request
  -->    26    csCode      word       ;always sendResponse
  -->    28    atpSocket   byte       ;socket number
  -->    29    atpFlags    byte       ;control information
  -->    30    addrBlock   long word  ;response destination
  -->    40    bdsPointer  pointer    ;pointer to response BDS
  -->    44    numOfBuffs  byte       ;number of response packets being sent
  -->    45    bdsSize     byte       ;BDS size in elements
  -->    46    transID     word       ;transaction ID
```

SendResponse sends a response to a socket. If the response was part of an exactly-once transaction, userData will contain the user bytes from the TRel packet. ATPSocket contains the socket number from which the response should be sent. The end-of-message flag in atpFlags should be set if the response contains the final packet in a transaction composed of a group of packets and the number of responses is less than requested. AddrBlock indicates the address of the socket to which the response should be sent. BDSPointer points to a response BDS containing room for the maximum number of responses to be sent; bdsSize contains this maximum number. NumOfBuffs contains the number of response packets to be

sent in this call; you may wish to make AddResponse calls to complete the response. TransID indicates the transaction ID of the associated request.

During exactly-once transactions, SendResponse doesn't complete until either a TRel packet is received from the socket that made the request, or the retry count is exceeded.

```
Result codes    noErr         No error
                badATPSkt     Bad responding socket
                noRelErr      No release received
                noDataArea    Too many outstanding ATP calls
                badBuffNum    Sequence number out of rangeAddResponse function
```

```
Parameter block
  -->    18    userData    long word  ;user bytes
  -->    26    csCode      word       ;always addResponse
  -->    28    atpSocket   byte       ;socket number
  -->    29    atpFlags    byte       ;control information
  -->    30    addrBlock   long word  ;response destination
  -->    34    reqLength   word       ;response size
  -->    36    reqPointer  pointer    ;pointer to response
  -->    44    rspNum      byte       ;sequence number
  -->    46    transID     word       ;transaction ID
```

AddResponse sends an additional response packet to a socket that has already been sent the initial part of a response via SendResponse. UserData contains the four user bytes. ATPSocket contains the socket number from which the response should be sent. The end-of-message flag in atpFlags should be set if this response packet is the final packet in a transaction composed of a group of packets and the number of responses is less than requested. AddrBlock indicates the socket to which the response should be sent. ReqLength and reqPointer contain the size (in bytes) and location of the response to send; rspNum indicates the sequence number of the response (in the range 0 to 7). TransID must contain the transaction ID.

```
Warning:  If the transaction is part of an exactly-once transaction, the
          buffer used in the AddResponse call must not be altered or
          released until the corresponding SendResponse call has completed.
```

```
Result codes    noErr         No error
                badATPSkt     Bad responding socket
                noSendResp    AddResponse issued before SendResponse
                badBuffNum    Sequence number out of range
                noDataArea    Too many outstanding ATP calls
```

```
RelTCB function

Parameter block
  -->    26    csCode     word       ;always relTCB
  -->    30    addrBlock  long word  ;destination of request
  -->    46    transID    word       ;transaction ID of request
```

RelTCB dequeues the specified SendRequest call and returns the result code reqAborted for the aborted call. The transaction ID can be obtained from the

reqTID field of the SendRequest queue entry; see the description of SendRequest for details.

```
Result codes    noErr         No error
                cbNotFound    ATP control block not found
                noDataArea    Too many outstanding ATP calls
```

RelRspCB function

```
Parameter block
  -->    26    csCode     word      ;always relRspCB
  -->    28    atpSocket  byte      ;socket number that request was received on
  -->    30    addrBlock  long word ;source of request
  -->    46    transID    word      ;transaction ID of request
```

In an exactly-once transaction, RelRspCB cancels the specified SendResponse, without waiting for the release timer to expire or a TRel packet to be received. No error is returned for the SendResponse call. Whan called to cancel a transaction that isn't using exactly-once service, RelRspCB returns cbNotFound. The transaction ID can be obtained from the reqTID field of the SendResponse queue entry; see the description of SendResponse for details.

```
Result codes    noErr         No error
                cbNotFound    ATP control block not found
```

Name-Binding Protocol

Data Structures

The first two bytes in the NBP header (Figure 13) indicate the type of the packet, the number of tuples in the packet, and an NBP packet identifier. You can use the following global constants to access these bytes:

```
  nbpControl  .EQU    0    ;packet type
  nbpTCount   .EQU    0    ;tuple count
  nbpID       .EQU    1    ;packet identifier
  nbpTuple    .EQU    2    ;start of first tuple
```

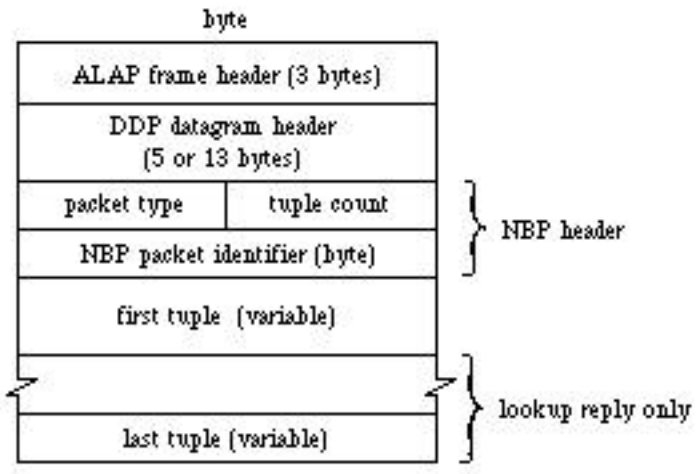

Figure 13–NBP Packet

Figure 13–NBP Packet

NBP packets are identified by the following DDP protocol type:

```
  nbp         .EQU    2
```

NBP uses the following global constants in the nbpControl field to identify NBP packets:

```
  brRq        .EQU    1    ;broadcast request
  lkUp        .EQU    2    ;lookup request
  lkUpReply   .EQU    3    ;lookup reply
```

NBP entities are identified by internet address in the form shown in Figure 14 below. Entities are also identified by tuples, which include both an internet address and an entity name. You can use the following global constants to access information in tuples:

```
  tupleNet    .EQU    0    ;network number
  tupleNode   .EQU    2    ;node ID
  tupleSkt    .EQU    3    ;socket number
  tupleEnum   .EQU    4    ;used internally
  tupleName   .EQU    5    ;entity name
```

The meta-characters in an entity name can be identified with the following global constants:

```
  equals      .EQU    '='    ;"wild-card" meta-character
  star        .EQU    '*'    ;"this zone" meta-character
```

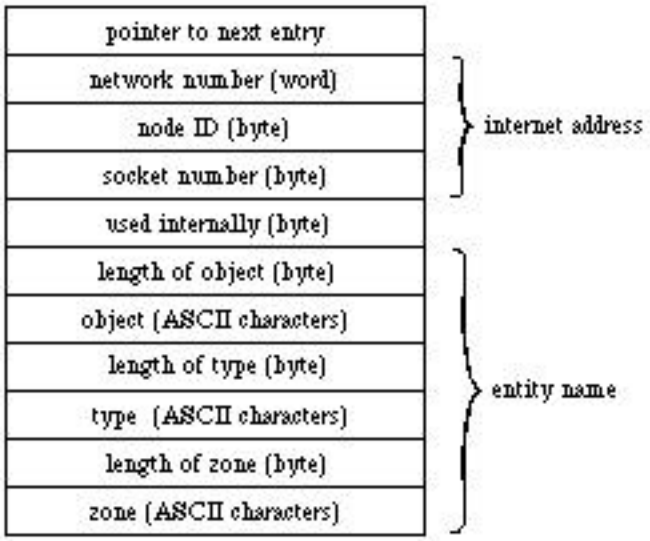

Figure 14–Names Table Entry

Figure 14–Names Table Entry

The maximum number of tuples in an NBP packet is given by the following global constant:

```
tupleMax    .EQU    15
```

Entity names are mapped to sockets via the names table. Each entry in the names table has the structure shown in Figure 14.

You can use the following global constants to access some of the elements of a names table entry:

```
ntLink      .EQU    0    ;pointer to next entry
ntTuple     .EQU    4    ;tuple
ntSocket    .EQU    7    ;socket number
ntEntity    .EQU    9    ;entity name
```

The socket number of the names information socket is given by the following global constant:

```
nis         .EQU    2
```

Using NBP

On a Macintosh 128K, before calling any other NBP routines, call the LoadNBP function, which reads the NBP code from the system resource file into the application heap. (The NBP code is part of the .MPP driver, which has a driver reference number of –10.) When you're finished with NBP and want to reclaim the space its code occupies, call UnloadNBP. On a Macintosh 512K or XL, the NBP code is read in when the .MPP driver is loaded.

Warning: When an application starts up, the application heap is

reinitialized; on a Macintosh 128K, this means that the NBP code is lost (and must be reloaded by the next application).

When an entity wants to communicate via an AppleTalk network, it should call RegisterName to place its name and internet address in the names table. When an entity no longer wants to communicate on the network, or is being shut down, it should call RemoveName to remove its entry from the names table.

To determine the address of an entity you know only by name, call LookupName, which returns a list of all entities with the name you specify. If you already know the address of an entity, and want only to confirm that it still exists, call ConfirmName. ConfirmName is more efficient than LookupName in terms of network traffic.

NBP Routines

RegisterName function

```
Parameter block
  -->    26    csCode      word     ;always registerName
  -->    28    interval    byte     ;retry interval
  <->    29    count       byte     ;retry count
  -->    30    ntQElPtr    pointer  ;names table element pointer
  -->    34    verifyFlag  byte     ;set if verify needed
```

RegisterName adds the name and address of an entity to the node's names table. NTQElPtr points to a names table entry containing the entity's name and internet address (in the form shown in Figure 14 above). Meta-characters aren't allowed in the object and type fields of the entity name; the zone field, however, must contain the meta-character "*". If verifyFlag is TRUE, RegisterName checks on the network to see if the name is already in use, and returns a result code of nbpDuplicate if so. Interval and count contain the retry interval in eight-tick units and the retry count. When a retry is made, the count field is modified.

•••Click on the X-Ref button, and refer to Technical Note #225.•••

Warning: The names table entry passed to RegisterName remains the property of NBP until removed from the names table. Don't attempt to remove or modify it. If you've allocated memory using a NewHandle call, you must lock it as long as the name is registered.

Warning: VerifyFlag should normally be set before calling RegisterName.

```
Result codes    noErr           No error
                nbpDuplicate    Duplicate name already exists
                nbpNISErr       Error opening names information socket
```

LookupName function

```
Parameter block
  -->    26    csCode       word     ;always lookupName
  -->    28    interval     byte     ;retry interval
  <->    29    count        byte     ;retry count
  -->    30    entityPtr    pointer  ;pointer to entity name
```

```
  -->    34    retBuffPtr   pointer  ;pointer to buffer
  -->    38    retBuffSize  word     ;buffer size in bytes
  -->    40    maxToGet     word     ;matches to get
  <--    42    numGotten    word     ;matches found
```

LookupName returns the addresses of all entities with a specified name. EntityPtr points to the entity's name (in the form shown in Figure 14 above). Meta-characters are allowed in the entity name. RetBuffPtr and retBuffSize contain the location and size of an area of memory in which the tuples describing the entity names and their corresponding addresses should be returned. MaxToGet indicates the maximum number of matching names to find addresses for; the actual number of addresses found is returned in numGotten. Interval and count contain the retry interval and the retry count. LookupName completes when either the number of matches is equal to or greater than maxToGet, or the retry count has been exceeded. The count field is decremented for each retransmission.

Note: NumGotten is first set to 0 and then incremented with each match found. You can test the value in this field, and can start examining the received addresses in the buffer while the lookup continues.

```
Result codes    noErr         No error
                nbpBuffOvr    Buffer overflow
```

ConfirmName function

Parameter block

```
  -->    26    csCode       word     ;always confirmName
  -->    28    interval     byte     ;retry interval
  <->    29    count        byte     ;retry count
  -->    30    entityPtr    pointer  ;pointer to entity name
  -->    34    confirmAddr  pointer  ;entity address
  <--    38    newSocket    byte     ;socket number
```

ConfirmName confirms that an entity known by name and address still exists (is still entered in the names directory). EntityPtr points to the entity's name (in the form shown in Figure 14 above). ConfirmAddr specifies the address to confirmed. No meta-characters are allowed in the entity name. Interval and count contain the retry interval and the retry count. The socket number of the entity is returned in newSocket. ConfirmName is more efficient than LookupName in terms of network traffic.

```
Result codes    noErr           No error
                nbpConfDiff     Name confirmed for different socket
                nbpNoConfirm    Name not confirmed
```

RemoveName function

Parameter block

```
  -->    26    csCode     word     ;always removeName
  -->    30    entityPtr  pointer  ;pointer to entity name
```

RemoveName removes an entity name from the names table of the given entity's node.

```
Result codes    noErr          No error
                nbpNotFound    Name not found
```

LoadNBP function

```
Parameter block
  -->    26    csCode  word  ;always loadNBP
```

On a Macintosh 128K, LoadNBP reads the NBP code from the system resource file into the application heap; on a Macintosh 512K or XL it has no effect.

```
Result codes    noErr    No error
```

UnloadNBP function

```
Parameter block
  -->    26    csCode  word  ;always unloadNBP
```

On a Macintosh 128K, UnloadNBP makes the NBP code purgeable; the space isn't actually released by the Memory Manager until necessary. On a Macintosh 512K or XL, UnloadNBP has no effect.

```
Result codes    noErr    No error
```

EXTENDED PROTOCOL PACKAGE DRIVER

The Extended Protocol Package (XPP) driver is intended to implement several AppleTalk communication protocols in the same package for ease of use. The .XPP driver currently consists of two modules that operate on two levels: the low-level module implements the workstation side of AppleTalk Session Protocol, and the high-level module implements a small portion of the workstation side of the AppleTalk Filing Protocol.

This driver adds functionality to the AppleTalk manager by providing services additional to those provided in the .MPP and .ATP drivers. Figure 2 shows the Macintosh AppleTalk drivers and the protocols accessible through each driver.

The .XPP driver maps an AFP call from the client workstation into one or more ASP calls. .XPP provides one client-level call for AFP.

The implementation of AFP in the .XPP driver is very limited. Most calls are a very simple one-to-one mapping from an AFP call to an ASP command without any interpretation of the syntax of the AFP command by the .XPP driver. Refer to the "Mapping AFP Commands" section of this chapter for further information.

Version

The .XPP driver supports ASP Version (hex) $100, as described in Inside AppleTalk.

Error Reporting

Errors are returned by the .XPP driver in the ioResult field of the Device Manager Control calls.

The error conditions reported by the .XPP driver may represent the unsuccessful completion of a routine in more than just one process involved in the interaction of the session. System-level, .XPP driver, AppleTalk, and server errors can all turn up in the ioResult field.

AFP calls return codes indicating the unsuccessful completion of AFP commands in the Command Result field of the parameter block (described below).

An application using the .XPP driver should respond appropriately to error conditions reported from the different parts of the interaction. As shown in Figure 3, the following errors can be returned in the ioResult field:

1. System-level errors

   System errors returned by the .XPP driver indicate such conditions as the driver not being open or a specific system call not being

supported. For a complete list of result codes returned by the Macintosh system software, refer to Appendix A.

2. XPP errors (for example, "Session not opened")

   The .XPP driver can also return errors resulting from its own activity (for example, the referenced session isn't open). The possible .XPP driver errors returned are listed in the .XPP driver results codes section with each function that can return the code.

3. AppleTalk Errors (returned from lower-level protocols)

   .XPP may also return errors from lower-level protocols (for example, "Socket not open"). Possible error conditions and codes are described elsewhere in this chapter.

4. An ASP-specific error could be returned from an ASP server in response to a failed OpenSession call. Errors of this type, returned by the server to the workstation, are documented both in Inside AppleTalk, section 11, "AppleTalk Session Protocol", and in the .XPP driver results code section of this chapter.

5. The AppleTalk Filing Protocol defines errors that are returned from the server to the workstation client. These errors are returned in the cmdResult field of the parameter block (error type 5 in Figure 15). This field is valid if no system-level error is returned by the call. Note that at the ASP level, the cmdResult field is client-defined data and may not be an error code.

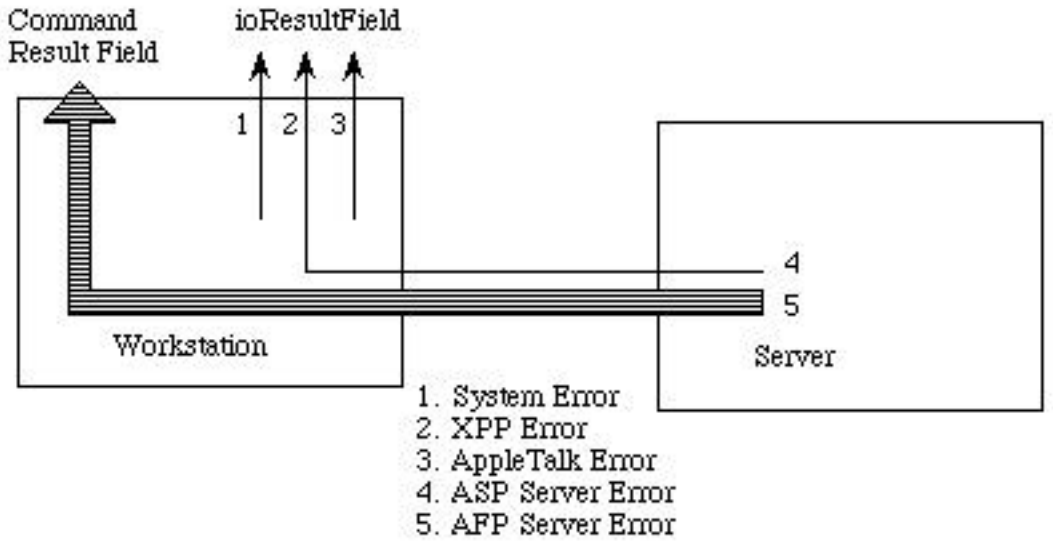

Figure 15–Error Reporting

Figure 15–Error Reporting

.XPP Driver Functions Overview

The paragraphs below describe the implementation of ASP in the .XPP driver. For more detailed information about ASP, refer to Inside AppleTalk, Section 11, "AppleTalk Session Protocol (ASP)".

Using AppleTalk Name Binding Protocol

A server wishing to advertise its service on the AppleTalk network calls ATP to open an ATP responding socket known as the session listening socket (SLS). The server then calls the Name Binding Protocol (NBP) to register a name on this socket. At this point, the server calls the server side of ASP to pass it the address of the SLS. Then, the server starts listening on the SLS for session opening requests coming over the network.

Opening and Closing Sessions

When a workstation wishes to access a server, the workstation must call NBP to discover the SLS for that server. Then the workstation calls ASP to open a session with that server.

After determining the SLS (address) of the server, the workstation client issues an OpenSession (or AFPLogin) call to open a session with that server. As a result of this call, ASP sends a special OpenSession packet (an ATP request) to the SLS; this packet carries the address of a workstation socket for use in the session. This socket is referred to as the workstation session socket (WSS). If the server is unable to set up the session, it returns an error. If the request is successful, the server returns no error, and the session is opened. The open session packet also contains a version number so that both ends can verify that they are speaking the same version of ASP.

The AbortOS function can be used to abort an outstanding OpenSession request before it has completed.

The workstation client closes the session by issuing a CloseSession (or AFPLogout). The CloseSession call aborts any calls that are active on the session and closes the session. The session can also be closed by the server or by ASP itself, such as when one end of the session fails. The CloseAll call (which should be used with care) aborts every session that the driver has active.

Session Maintenance

A session will remain open until it is explicitly terminated by the ASP client at either end or until one of the sessions ends, fails, or becomes unreachable.

Commands on an Open Session

Once a session has been opened, the workstation client can send a sequence of commands over the session to the server end. The commands are delivered in the same order as they are issued at the workstation end, and replies to the commands are returned to the workstation end.

Three types of commands can be made on an open session. These commands are UserCommand, UserWrite, and AFPCall functions described in the following paragraphs.

UserCommand calls are similar to ATP requests. The workstation client sends a command (included in a variable size command block) to the server client requesting it to perform a particular function and send back a variable size command reply. Examples of such commands vary from a request to open a particular file on a file server, to reading a certain range of bytes from a device. In the first case, a small amount of reply data is returned; in the second case a multiple-packet reply might be generated.

The .XPP driver does not interpret the command block or in any way participate in executing the command's function. It simply conveys the command block, included in a higher-level format, to the server end of the session, and returns the command reply to the workstation-end client. The command reply consists of a four-byte command result and a variable size command reply block.

UserWrite allows the workstation to convey blocks of data to the server. UserWrite is used to transfer a variable size block of data to the server end of the session and to receive a reply.

The AFPCall function provides a mechanism for passing an AFP command to the server end of an open session and receiving a reply. The first byte of the AFPCall command buffer contains the code for the AFP command that is to be passed to the server for execution. Most AFP calls are implemented through a very simple one-to-one mapping that takes the call and makes an ASP command out of it.

The AFPCall function can have one of four different, but very similar, formats.

Getting Server Status Information

ASP provides a service to allow its workstation clients to obtain a block of service status information from a server without the need for opening a session. The GetStatus function returns a status block from the server identified by the indicated address. ASP does not impose any structure on the status block. This structure is defined by the protocol above ASP.

Attention Mechanism

Attentions are defined in ASP as a way for the server to alert the workstation of some event or critical piece of information. The ASP OpenSession and AFPLogin calls include a pointer to an attention routine in their parameter blocks. This attention routine is called by the .XPP driver when it receives an attention from the server and also when the session is closing as described below.

In addition, upon receiving an OpenSession call or AFPLogin call, the .XPP driver sets the first two bytes of the session control block (SCB) to zero. When the .XPP driver receives an attention, the first two bytes of the SCB are set to the attention bytes from the packet (which are always nonzero).

Note: A higher-level language such as Pascal may not wish to have a low-level attention routine called. A Pascal program can poll the attention bytes,

and if they are ever nonzero, the program will know that an attention has come in. (It would then set the attention bytes back to zero.) Of course, two or more attentions could be received between successive polls, and only the last one would be recorded.

The .XPP driver also calls the attention routine when the session is closed by either the server, workstation, or ASP itself (if the ASP session times out). In these cases, the attention bytes in the SCB are unchanged.

The Attention Routine

The attention routine is called at interrupt level and must observe interrupt conventions. Specifically, the interrupt routine can change registers A0 through A3 and D0 through D3 and it must not make any Memory Manager calls.

It will be called with

- D0 (word) equal to the SessRefnum for that session (see OpenSession Function)
- D1 (word) equal to the attention bytes passed by the server (or zero if the session is closing)

Return with an RTS (return from subroutine) to resume normal execution.

The next section describes the calls that can be made to the .XPP driver.

CALLING THE .XPP DRIVER

This section describes how to use the .XPP driver and how to call the .XPP driver routines from assembly language and Pascal.

Using XPP

The .XPP driver implements the workstation side of ASP and provides a mechanism for the workstation to send AppleTalk Filing Protocol (AFP) commands to the server.

Allocating Memory

Every call to the .XPP driver requires the caller to pass in whatever memory is needed by the driver for the call, generally at the end of the queue element. When a session is opened, the memory required for maintenance of that session (that is, the Session Control Block) is also passed in.

For standard Device Manager calls, a queue element of a specific size equal to IOQElSize is allocated. When issuing many calls to XPP, it is the caller's responsibility to allocate a queue element that is large enough to accommodate the .XPP driver's requirements for executing that call, as defined below. Once allocated, that memory can't be modified until the call completes.

Opening the .XPP Driver

To open the .XPP driver, issue a Device Manager Open call. (Refer to the Device Manager chapter.) The name of the .XPP driver is '.XPP'. The original Macintosh ROMs require that .XPP be opened only once. With new ROMs, the .XPP unit number can always be obtained through an Open call. With old ROMs only, the .XPP unit number must be hard coded to XPPUnitNum (40) since only one Open call can be issued to the driver.

The .XPP driver cannot be opened unless AppleTalk is open. The application must ensure that the .MPP and .ATP drivers are opened, as described earlier in this chapter.

The xppLoaded bit (bit 5) in the PortBUse byte in low memory indicates whether or not the .XPP driver is open.

Example

The following is an example of the procedure an application might use to open the .XPP driver.

```
;    Routine: OpenXPP
;
;        Open the .XPP driver and return the driver refNum for it.
;
```

```
;        Exit:    D0 = error code (ccr's set)
;                 D1 = XPP driver refNum (if no errors)
;
;        All other registers preserved
;
xppUnitNum    EQU    40                ;default XPP driver number
xppTfRNum     EQU    -(xppUnitNum+1)   ;default XPP driver refNum

OpenXPP
    MOVE.L    A0-A1/D2,-(SP)           ;save registers
    MOVE      ROM85,D0                 ;check ROM type byte
    BPL.S     @10                      ;branch if >=128K ROMs
    BTST      #xppLoadedBit,PortBUse   ;is the XPP driver open already?
    BEQ.S     @10                      ;if not open, then branch to Open code
    MOVE      #xppTfRNum,D1            ;else use this as driver refnum
    MOVEQ     #0,D0                    ;set noErr
    BRA.S     @90                      ;and exit
;
; XPP driver not open. Make an _Open call to it. If using a 128K
; ROM machine and the driver is already open, we will make another
; Open call to it just so we get the correct driver refNum.
;
@10 SUB       #ioQElSize,SP            ;allocate temporary param block
    MOVE.L    SP,A0                    ;A0 -> param block
    LEA       XPPName, A1              ;A1 -> XPP (ASP/AFP) driver name
    MOVE.L    A1,ioFileName(A0)        ;driver name into param block
    CLR.B     ioPermssn(A0)            ;clear permissions byte
    _Open
    MOVE      ioRefNum(A0),D1          ;D1=driver refNum (invalid if error)
    ADD       #ioQElSize,SP            ;deallocate temp param block
@90 MOVE.L    (SP)+,A0-A1/D2           ;restore registers
    TST       D0                       ;error? (set ccr's)
    RTS

XPPName  DC.B  4                       ;length of string
         DC.B  '.XPP'                  ;driver name
```

From Pascal, XPP can be opened through the OpenXPP call, which returns the driver's reference number:

```
FUNCTION OpenXPP (VAR xppRefnum: INTEGER) : OSErr;
```

Open Errors

Errors returned when calling the Device Manager Open routine if the function does not execute properly include the following:

- errors returned by System
- portInUse is returned if the AppleTalk port is in use by a driver other than AppleTalk or if AppleTalk is not open.

Closing the .XPP Driver

To close the .XPP driver, call the Device Manager Close routine.

Warning: There is generally no reason to close the driver. Use this call sparingly, if at all. This call should generally be used only by system-level applications.

Close Errors

Errors returned when calling the Device Manager Close routine if the function does not execute properly include the following:

- errors returned by System
- closeErr (new ROMs only) is returned if you try to close the driver and there are sessions active through that driver. When sessions are active, closeErr is returned and the driver remains open.
- on old ROMs the driver is closed whether or not sessions are active and no error is returned. Results are unpredictable if sessions are still active.

Session Control Block

The session control block (SCB) is a nonrelocatable block of data passed by the caller to XPP upon session opening. XPP reserves this block for use in maintaining an open session. The SCB size is defined by the constant scbMemSize. The SCB is a locked block, and as long as the session is open, the SCB cannot be modified in any way by the application. There is one SCB for each open session. This block can be reused once a CloseSess call is issued and completed for that session or when the session is indicated as closed.

How to Access the .XPP Driver

This section contains information for programmers using Pascal and assembly-language routines.

All .XPP driver routines can be executed either synchronously (meaning that the application can't continue until the routine is completed) or asynchronously (meaning that the application is free to perform other tasks while the routine is executing).

XPP calls are made from Pascal in the same manner as MPP and ATP calls, with the exception that when making XPP calls the caller must set the XPP driver's refnum. This refnum is returned in the XPPOpen call's parameter block.

A Pascal variant record has been defined for all XPP calls. This parameter block is detailed in the ".XPP Driver Parameter Block Record" section below. The first four fields (which are the same for all calls) are automatically filled in by the device manager. The csCode field is automatically filled in by Pascal, depending on which call is being made. The caller must, however, set the ioRefnum field to XPP's reference number, as returned in the OpenXPP call. The ioVRefnum field is unused.

Note that the parameter block is defined so as to be the maximum size used by any call. Different calls take different size parameter blocks, each call requiring a certain minimum size. Callers are free to abbreviate the parameter block where appropriate.

General

With each routine, a list of the parameter block fields used by the call is also given. All routines are invoked by Device Manager Control calls with the csCode field equal to the code corresponding to the function being called. The number next to each field name indicates the byte offset of the field from the start of the parameter block pointed to by A0; only assembly-language programmers need to be concerned with it. An arrow next to each parameter name indicates whether it's an input, output, or input/output parameter:

Arrow	Meaning
<--	Parameter is passed
<--	Parameter is returned
<->	Parameter is passed and returned

All Device Manager Control calls return an integer result code in the ioResult field. Each routine description lists all the applicable result codes, along with a short description of what the result code means. Refer to the section "XPP Driver Result Codes" for an alphabetical list of result codes returned by the .XPP driver.

Each routine description includes a Pascal form of the call. Pascal calls to the .XPP Driver are of the form:

```
FUNCTION XPPCall (paramBlock: XPPParmBlkPtr,async: BOOLEAN) : OSErr;
```

XPPCall is the name of the routine.

The parameter paramBlock points to the actual I/O queue element used in the _Control call, filled in by the caller with the parameters of the routine.

The parameter async indicates whether or not the call should be made asynchronously. If async is TRUE, the call is executed asynchronously; otherwise the call is executed synchronously.

The routine returns a result code of type OSErr.

.XPP Driver Parameter Block Record

```
XPPParamBlock = PACKED RECORD
    qLink:          QElemPtr;  {next queue entry}
    qType:          INTEGER;   {queue type}
    ioTrap:         INTEGER;   {routine trap}
    ioCmdAddr:      Ptr;       {routine address}
    ioCompletion:   ProcPtr;   {completion routine}
    ioResult:       OSErr;     {result code}
    cmdResult:      LONGINT;   {command result (ATP user bytes) [long]}
    ioVRefNum:      INTEGER;   {volume reference or drive number)
    ioRefNum:       INTEGER;   {driver reference number)
    csCode:         INTEGER;   {Call command code}
    CASE XPPPrmBlkType OF
      ASPAbortPrm:
        (abortSCBPtr:    Ptr);      {SCB pointer for AbortOS [long]}
      ASPSizeBlk:
```

```
    (aspMaxCmdSize:  INTEGER;   {for SPGetParms [word]
    aspQuantumSize:  INTEGER;   {for SPGetParms [word]}
    numSesss:        INTEGER);  {for SPGetParms [word]}
  XPPPrmBlk:
    (sessRefnum:     INTEGER;   {offset to session refnum [word]}
    aspTimeout:      Byte;      {timeout for ATP [byte]}
    aspRetry:        Byte;      {retry count for ATP [byte]}
    CASE XPPSubPrmType OF
      ASPOpenPrm:
        (serverAddr:    AddrBlock;  {server address block [longword]}
        scbPointer:     Ptr;        {SCB pointer [longword]}
        attnRoutine:    Ptr);       {attention routine pointer [long]}
      ASPSubPrm:
        (cbSize:        INTEGER;    {command block size [word]}
        cbPtr:          Ptr;        {command block pointer [long]}
        rbSize:         INTEGER;    {reply buffer size [word]}
        rbPtr:          Ptr;        {reply buffer pointer [long]}
        CASE XPPEndPrmType OF
          AFPLoginPrm:
            (afpAddrBlock:     AddrBlock;    {address block in}
                                             { AFPlogin [long]}
            afpSCBPtr:         Ptr;          {SCB pointer in }
                                             { AFPlogin [long]}
            afpAttnRoutine:    Ptr);         {attn routine pointer }
                                             { in AFPlogin}
          ASPEndPrm:
            (wdSize:           INTEGER;      {write data size [word]}
            wdPtr:             Ptr;          {write data pointer [long]}
            ccbStart:          ARRAY[0..295] OF Byte)));   {CCB memory }
                                                           { for driver}
{Write max size(CCB) = 296; all other calls = 150}
END;
```

AppleTalk Session Protocol Functions

This section contains descriptions of the .XPP driver functions that you can call. Each function description shows the required parameter block fields, their offsets within the parameter block and a brief definition of the field. Possible result codes are also described.

Note on Result Codes

An important distinction exists between the aspParamErr and aspSessClose result codes that may be returned by the .XPP driver.

When the driver returns aspParamErr to a call that takes as an input a session reference number, the session reference number does not relate to a valid open session. There could be several reasons for this, such as the workstation or server end closed the session or the server end of the session died.

The aspSessClosed result code indicates that even though the session reference number relates to a valid session, that particular session is in the process of closing down (although the session is not yet closed).

```
FUNCTION ASPOpenSession (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;

Parameter block
  -->    26    csCode       word       Always ASPOpenSess
  -->    28    sessRefnum   word       Session reference number
  -->    30    aspTimeout   byte       Retry interval in seconds
  -->    31    aspRetry     byte       Number of retries
  -->    32    serverAddr   long word  Server socket address
  -->    36    scbPointer   pointer    Pointer to session control block
  -->    40    attnRoutine  pointer    Pointer to attention routine
```

ASPOpenSession initiates (opens) a session between the workstation and a server. The required parameter block is shown above. A brief definition of the fields follows.

SessRefnum is a unique number identifying the open session between the workstation and the server. The SessRefnum is returned when the function completes successfully and is used in all calls to identify the session.

ASPTimeOut is the interval in seconds between retries of the open session request.

ASPRetry is the number of retries that will be attempted.

ServerAddr is the network identifier or address of the socket on which the server is listening.

SCBPointer points to a nonrelocatable block of data for the session control block (SCB) that the .XPP driver reserves for use in maintaining an open session. The SCB size is defined by the constant scbMemSize. The SCB is a locked block and as long as the session is open, the SCB cannot be modified in any way by the application. There is one SCB for each open session. This block can be reused when a CloseSess call is issued and completed for that session, or when the session is indicated as closed through return of aspParamErr as the result of a call for that session.

AttnRoutine is a pointer to a routine that is invoked if an attention from the server is received, or upon session closing. If this pointer is equal to zero, no attention routine will be invoked.

Result codes		
	aspNoMoreSess	Driver cannot support another session
	aspParamErr	Server returned bad (positive) error code
	aspNoServers	No servers at that address, or the server did not respond to the request
	reqAborted	OpenSess was aborted by an AbortOS
	aspBadVersNum	Server cannot support the offered version number
	aspServerBusy	Server cannot open another session

Note: The number of sessions that the driver is capable of supporting depends on the machine that the driver is running on.

```
FUNCTION ASPCloseSession (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;
```

```
Parameter block
  -->    26    csCode      word  Always ASPCloseSession
  -->    28    sessRefnum  word  Session reference number
```

ASPCloseSession closes the session identified by the sessRefnum returned in the ASPOpenSession call. ASPCloseSession aborts any calls that are active on the session, closes the session, and calls the attention routine, if any, with an attention code of zero (zero is invalid as a real attention code).

```
Result codes    aspParamErr      Parameter error, indicates an invalid
                                 session reference number
                aspSessClosed    Session already in process of closing
```

```
FUNCTION ASPAbortOS (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;

Parameter block
  -->    26    csCode            word     Always ASPAbortOS
  -->    28    abortSCBPointer   pointer  Pointer to session control block
```

ASPAbortOS aborts a pending (not yet completed) ASPOpenSession call. The aborted ASPOpenSession call will return a reqAborted error.

AbortSCBPointer points to the original SCB used in the the pending ASPOpenSession call.

```
Result codes    cbNotFound    SCB not found, no outstanding open session
                              to be aborted.
                              Pointer did not point to an open session SCB.
```

```
FUNCTION ASPGetParms (xParamBlock: XPPParmBlkPtr; async: BOOLEAN): OSErr;

Parameter block
  -->    26    csCode          word  Always ASPGetParms
  -->    28    aspMaxCmdSize   word  Maximum size of command block
  -->    30    aspQuantumSize  word  Maximum data size
  -->    32    numSesss        word  Number of sessions
```

ASPGetParms returns three ASP parameters. This call does not require an open session.

ASPMaxCmdSize is the maximum size of a command that can be sent to the server.

ASPQuantumSize is the maximum size of data that can be transferred to the server in a Write request or from the server in a command reply.

NumSess is the number of concurrent sessions supported by the driver.

```
FUNCTION ASPCloseAll (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;

Parameter block
  -->    26    csCode  word  Always ASPCloseAll
```

ASPCloseAll closes every session that the driver has active, aborting all active requests and invoking the attention routines where provided. This call should

be used carefully. ASPCloseAll can be used as a system level resource for making sure all sessions are closed prior to closing the driver.

```
FUNCTION ASPUserWrite (xParamBlock: XPPParmBlkPtr; async: BOOLEAN): OSErr;
```

Parameter block

-->	18	cmdResult	long word	ASP command result
-->	26	csCode	word	Always UserWrite
-->	28	sessRefnum	word	Session reference number
-->	30	aspTimeout	byte	Retry interval in seconds
-->	32	cbSize	word	Command block size
-->	34	cbPtr	pointer	Command block pointer
<->	38	rbSize	word	Reply buffer size and reply size
-->	40	rbPtr	pointer	Reply buffer pointer
<->	44	wdSize	word	Write data size
-->	46	wdPtr	pointer	Write data pointer
-->	50	ccbStart	record	Start of memory for CCB

ASPUserWrite transfers data on a session. ASPUserWrite is one of the two main calls that can be used to transfer data on an ASP session. The other call that performs a similar data transfer is ASPUserCommand described below. The ASPUserWrite command returns data in two different places. Four bytes of data are returned in the cmdResult field and a variable size reply buffer is also returned.

CmdResult is four bytes of data returned by the server.

SessRefnum is the session reference number returned in the ASPOpenSession call.

ASPTimeOut is the interval in seconds between retries of the call. Notice that there is no aspRetry field (retries are infinite). The command will be retried at the prescribed interval until completion or the session is closed.

CBSize is the size in bytes of the command data that is to be written on the session. The size of the command block must not exceed the value of aspMaxCmdSize returned by the ASPGetParms call. Note that this buffer is not the data to be written by the command but only the data of the command itself.

CBPtr points to the command data.

RBSize is passed and indicates the size of the reply buffer in bytes expected by the command. RBSize is also returned and indicates the size of the reply that was actually returned.

RBPtr points to the reply buffer.

WDSize is passed and indicates the size of the write data in bytes to be sent by the command. WDSize is also returned and indicates the size of the write data that was actually written.

WDPointer points to the write data buffer.

CCBStart is the start of the memory to be used by the .XPP driver for the command control block. The size of this block is equal to a maximum of 296

bytes. To determine the exact requirement, refer to the CCB Sizes section of this document.

Result codes	aspParamErr	Invalid session number, session has been closed
	aspSizeErr	Command block size is bigger than MaxCmdSize
	aspSessClosed	Session is closing
	aspBufTooSmall	Reply is bigger than response buffer; the buffer will be filled, data will be truncated

```
FUNCTION ASPUserCommand (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;
```

Parameter block

-->	18	cmdResult	long word	ASP command result
-->	26	csCode	word	Always ASPUserCommand
-->	28	sessRefnum	word	Session number
-->	30	aspTimeout	byte	Retry interval in seconds
-->	32	cbSize	word	Command block size
-->	34	cbPtr	pointer	Command block pointer
<->	38	rbSize	word	Reply buffer and reply size
-->	40	rbPtr	pointer	Reply buffer pointer
-->	50	ccbStart	record	Start of memory for CCB

ASPUserCommand is used to send a command to the server on a session.

SessRefnum is the session reference number returned in the ASPOpenSession call.

ASPTimeOut is the interval in seconds between retries of the call. Notice that there is no aspRetry field (retries are infinite). The command will be retried at the prescribed interval until completion or the session is closed.

CBSize is the size in bytes of the block of data that contains the command to be sent to the server on the session. The size of the command block must not exceed the value of aspMaxCmdSize returned by the ASPGetParms call.

CBPointer points to the block of data containing the command that is to be sent to the server on the session.

RBSize is passed and indicates the size of the reply buffer in bytes expected by the command. RBSize is also returned and indicates the size of the reply that was actually returned.

RBPtr points to the reply buffer.

CCBStart is the start of the memory to be used by the .XPP driver for the command control block. The size of this block is equal to a maximum of 150 bytes. To determine the exact requirement refer to the CCB Sizes section of this document.

Result codes	aspParamErr	Invalid session number, session has been closed
	aspSizeErr	Command block size is bigger than MaxCmdSize
	aspSessClosed	Session is closing
	aspBufTooSmall	Reply is bigger than response buffer;

			the buffer will be filled, data will be truncated

```
FUNCTION ASPGetStatus (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;
```

Parameter block

-->	26	csCode	word	Always ASPGetStatus
-->	30	aspTimeout	byte	Retry interval in seconds
-->	31	aspRetry	byte	Number of retries
-->	32	serverAddr	long word	Server socket address
<->	38	rbSize	word	Reply buffer and reply size
-->	40	rbPtr	pointer	Reply buffer pointer
-->	50	ccbStart	record	Start of memory for CCB

ASPGetStatus returns server status. This call is also used as GetServerInfo at the AFP level. This call is unique in that it transfers data over the network without having a session open. This call does not pass any data but requests that server status be returned.

ASPTimeOut is the interval in seconds between retries of the call.

ASPRetry is the number of retries that will be attempted.

ServerAddr is the network identifier or address of the socket on which the server is listening.

RBSize is passed and indicates the size of the reply buffer in bytes expected by the command. RBSize is also returned and indicates the size of the reply that was actually returned.

RBPtr points to the reply buffer.

CCBStart is the start of the memory to be used by the .XPP driver for the command control block. The size of this block is equal to a maximum of 150 bytes. To determine the exact requirement refer to the CCB Sizes section of this document.

Result codes	aspBufTooSmall	Reply is bigger than response buffer, or Replysize is bigger than ReplyBuffsize
	aspNoServer	No response from server at address used in call

AFP Implementation

The AFPCall function (called AFPCommand in Pascal) passes a command to an AFP server. The first byte of the AFPCall command buffer (the AFP command byte) must contain a valid AFP command code.

Note: Server information should be gotten through an ASPGetStatus call (described above). ASPGetStatus is equivalent to the AFPGetSrvrInfo. Making an AFP GetSrvrInfo call using AFPCommand results in an error.

Mapping AFP Commands

Most AFP calls are implemented by XPP through a very simple one-to-one mapping of an AFP call to an ASP call without interpretation or verification of the data.

The .XPP driver maps AFP command codes to ASP commands according to the following conventions:

AFP Command Code	Comment
$00	Invalid AFP command
$01–$BE (1–190)	Mapped to UserCommand (with the exceptions listed below)
$BF (191)	Mapped to UserCommand (Reserved for developers; will never be used by Apple)
$C0–$FD (192–253)	Mapped to UserWrite
$FE (254)	Mapped to UserWrite (will never be used by Apple)
$FF (255)	Invalid AFP command

The following AFP calls are exceptions to the above conventions:

AFP Command (Code/Decimal)	Comment
getSrvrInfo (15)	Mapped to ASPGetStatus (Use ASPGetStatus to make this call)
login (18)	Mapped to appropriate log-in dialog including ASPOpenSession call
loginCont (19)	Mapped to appropriate log-in dialog
logout (20)	Mapped to ASPCloseSession
write (33)	Mapped to ASPUserWrite

The following AFP calls can pass or return more data than can fit in quantumSize bytes (eight ATP response packets) and may be broken up by XPP into multiple ASP calls.

AFP Command (Code/Decimal)	Comment
read (27)	Can return up to the number of bytes indicated in reqCount
write (33)	Can pass up to the number of bytes indicated in reqCount

AFPCall Function

The AFPCall function can have one of the following command formats.

- General
- Login
- AFPWrite
- AFPRead

General Command Format

```
FUNCTION AFPCommand (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;
```

Parameter block

-->	18	cmdResult	long word	AFP command result
-->	26	csCode	word	Always AFPCall
-->	28	sessRefnum	word	Session reference number
-->	30	aspTimeout	byte	Retry interval in seconds
-->	32	cbSize	word	Command buffer size
-->	34	cbPtr	pointer	Command buffer
<->	38	rbSize	word	Reply buffer size and reply size
-->	40	rbPtr	pointer	Reply buffer pointer
<->	44	wdSize	word	Write data size
-->	46	wdPtr	pointer	Write data pointer
-->	50	ccbStart	record	Start of memory for CCB

The general command format for the AFPCall function passes an AFP command to the server. This format is used for all AFP calls except AFPLogin, AFPRead, and AFPWrite. Note that from Pascal this call is referred to as AFPCommand.

CmdResult is four bytes of data returned from the server containing an indication of the result of the AFP command.

SessRefnum is the session reference number returned in the AFPLogin call.

ASPTimeOut is the interval in seconds between retries of the call by the driver.

CBSize is the size in bytes of the block of data that contains the command to be sent to the server on the session. The size of the command block must not exceed the value of aspMaxCmdSize returned by the ASPGetParms call.

CBPtr points to start of the block of data (command block) containing the command that is to be sent to the server on the session. The first byte of the command block must contain the AFP command byte. Subsequent bytes in the command buffer contain the parameters associated with the command as defined in the AFP document.

RBSize is passed and indicates the size of the reply buffer in bytes expected by the command. RBSize is also returned and indicates the size of the reply that was actually returned.

RBPtr points to the reply buffer.

WDSize is the size of data to be written to the server (only used if the command is one that is mapped to an ASPUserWrite).

WDPtr points to the write data buffer (only used if the command is one that is mapped to an ASPUserWrite).

CCBStart is the start of the memory to be used by the .XPP driver for the command control block. The size of this block is equal to a maximum of 296 bytes. To determine the exact requirement refer to the CCB Sizes section of this document.

Result codes	aspParamErr	Invalid session number; session has been closed
	aspSizeErr	Command block size is bigger than MaxCmdSize
	aspSessClosed	Session is closing

aspBufTooSmall	Reply is bigger than response buffer or buffer will be filled, data will be truncated
afpParmError	AFP command block size is equal to zero. This error will also be returned if the command byte in the command block is equal to 0 or $FF (255) or GetSrvrStatus (15).

Login Command Format

The AFP login command executes a series of AFP operations as defined in the AFP Draft Proposal. For further information, refer to the AFP document.

```
FUNCTION AFPCommand (xParamBlock: XPPParmBlkPtr; async: BOOLEAN): OSErr;
```

Parameter block

-->	18	cmdResult	long word	AFP command result
-->	26	csCode	word	Always AFPCall
-->	28	sessRefnum	word	Session reference number
-->	30	aspTimeout	byte	Retry interval in seconds
-->	31	aspRetry	byte	Number of retries
-->	32	cbSize	word	Command buffer size
-->	34	cbPtr	pointer	Command buffer
<->	38	rbSize	word	Reply buffer size and reply size
-->	40	rbPtr	pointer	Reply buffer pointer
-->	44	afpAddrBlock	long word	Server address block
<->	48	afpSCBPtr	pointer	SCB pointer
<->	52	afpAttnRoutine	pointer	Attention routine pointer
-->	50	ccbStart	record	Start of command control block

CmdResult is four bytes of data returned from the server containing an indication of the result of the AFP command.

SessRefnum is the session reference number (returned by the AFPLogin call).

ASPTimeOut is the interval in seconds between retries of the call.

ASPRetry is the number of retries that will be attempted.

CBSize is the size in bytes of the block data that contains the command to be sent to the server on the session. The size of the command block must not exceed the value of aspMaxCmdSize returned by the ASPGetParms call.

CBPtr points to the block of data (command block) containing the AFP login command that is to be sent to the server on the session. The first byte of the command block must be the AFP login command byte. Subsequent bytes in the command buffer contain the parameters associated with the command.

RBSize is passed and indicates the size of the reply buffer in bytes expected by the command. RBSize is also returned and indicates the size of the reply that was actually returned.

RBPtr points to the reply buffer.

AFPServerAddr is the network identifier or address of the socket on which the server is listening.

AFPSCBPointer points to a locked block of data for the session control block (SCB). The SCB size is defined by scbMemSize. The SCB is a locked block, and as long as the session is open, the SCB cannot be modified in any way by the application. There is one SCB for each open session.

AFPAttnRoutine is a pointer to a routine that is invoked if an attention from the server is received. When afpAttnRoutine is equal to zero, no attention routine will be invoked.

CCBStart is the start of the memory to be used by the .XPP driver for the command control block. The size of this block is equal to a maximum of 150 bytes. To determine the exact requirement refer to the CCB Sizes section later in this chapter.

Note: In the parameter block, the afpSCBPointer and the afpAttnRoutine fields overlap with the start of the CCB and are modified by the call.

Result codes	aspSizeErr	Command block size is bigger than MaxCmdSize
	aspBufTooSmall	Reply is bigger than response buffer or buffer will be filled, data will be truncated
	aspNoServer	Server not responding
	aspServerBusy	Server cannot open another session
	aspBadVersNum	Server cannot support the offered ASP version number
	aspNoMoreSess	Driver cannot support another session.AFPWrite

Command Format

The AFPWrite and AFPRead command formats allow the calling application to make AFP-level calls that read or write a data block that is larger than a single ASP-level call is capable of reading or writing. The maximum number of bytes of data that can be read or written at the ASP level is equal to quantumSize.

```
FUNCTION AFPCommand (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;
```

Parameter block

-->	18	cmdResult	long word	AFP command result
-->	26	csCode	word	Always AFPCall
-->	28	sessRefnum	word	Session number
-->	30	aspTimeout	byte	Retry interval in seconds
-->	32	cbSize	word	Command buffer size
-->	34	cbPtr	pointer	Command buffer
<->	38	rbSize	word	Reply buffer size and reply size
-->	40	rbPtr	pointer	Reply buffer pointer
-->	44	wdSize	word	(used internally)
<->	46	wdPtr	pointer	Write data pointer (updated)
-->	50	ccbStart	record	Start of memory for CCB

CmdResult is four bytes of data returned from the server containing an indication of the result of the AFP command.

SessRefnum is the session reference number returned in the AFPLogin call.

ASPTimeOut is the interval in seconds between retries of the call.

CBSize is the size in bytes of the block data that contains the command to be sent to the server on the session. The size of the command block must not exceed the value of aspMaxCmdSize returned by the aspGetParms call.

CBPtr points to the block of data (see command block structure below) containing the AFP write command that is to be sent to the server on the session. The first byte of the Command Block must contain the AFP write command byte.

RBSize is passed and indicates the size of the reply buffer in bytes expected by the command. RBSize is also returned and indicates the size of the reply that was actually returned.

RBPtr points to the reply buffer.

WDSize is used internally.

Note: This command does not pass the write data size in the queue element but in the command buffer. XPP will look for the size in that buffer.

WDPtr is a pointer to the block of data to be written. Note that this field will be updated by XPP as it proceeds and will always point to that section of the data which XPP is currently writing.

CCBStart is the start of the memory to be used by the XPP driver for the command control block. The size of this block is equal to a maximum of 296 bytes. To determine the exact requirement refer to the CCB Sizes section later in this chapter.

Command Block Structure: The AFP write command passes several arguments to XPP in the command buffer itself. The byte offsets are relative to the location pointed to by cbPtr.

-->	0	cmdByte	byte	AFP call command byte
-->	1	startEndFlag	byte	Start/end Flag
<->	4	rwOffset	long word	Offset within fork to write
<->	8	reqCount	long word	Requested count

CmdByte is the AFP call command byte and must contain the AFP write command code.

StartEndFlag is a one-bit flag (the high bit of the byte) indicating whether the rwOffset field is relative to the beginning or the end of the fork (all other bits are zero).

0 = relative to the beginning of the fork
1 = relative to the end of the fork

RWOffset is the byte offset within the fork at which the write is to begin.

ReqCount indicates the size of the data to be written and is returned as the actual size written.

The rwOffset and reqCount fields are modified by XPP as the write proceeds and will always indicate the current value of these fields.

The Pascal structure of the AFP command buffer follows:

```
AFPCommandBlock = PACKED RECORD
                    cmdByte:       Byte;
                    startEndFlag:  Byte;
                    forkRefNum:    INTEGER;    {used by server}
                    rwOffset:      LONGINT;
                    reqCount:      LONGINT;
                    newLineFlag:   Byte;       {unused by write}
                    newLineChar:   CHAR;       {unused by write}
                  END;
```

Result codes	aspParamErr	Invalid session number
	aspSizeErr	Command block size is bigger than MaxCmdSize
	aspSessClosed	Session is closing
	aspBufTooSmall	Reply is bigger than response buffer

AFPRead Command Format

The AFPWrite and AFPRead command formats allow the calling application to make AFP-level calls that read or write a data block that is larger than a single ASP-level call is capable of reading or writing. The maximum number of bytes of data that can be read or written at the ASP level is equal to quantumSize.

```
FUNCTION AFPCommand (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;
```

Parameter block

-->	18	cmdResult	long word	ASP command result
-->	26	csCode	word	Always AFPCall
-->	28	sessRefnum	word	Session number
-->	30	aspTimeout	byte	Retry interval in seconds
-->	32	cbSize	word	Command buffer size
-->	34	cbPtr	pointer	Command buffer
-->	38	rbSize	word	Used internally
<->	40	rbPtr	pointer	Reply buffer pointer (updated)
-->	50	ccbStart	record	Start of memory for CCB

CmdResult is four bytes of data returned from the server containing an indication of the result of the AFP command.

SessRefnum is the session reference number returned in the AFPLogin call.

ASPTimeOut is the interval in seconds between retries of the call.

CBSize is the size in bytes of the block data that contains the command to be sent to the server on the session. The size of the command block must not exceed the value of aspMaxCmdSize returned by the GetParms call.

CBPtr points to the block of data (command block) containing the AFP read command that is to be sent to the server on the session. The first byte of the command block must contain the AFP read command byte. The command block structure is shown below.

RBSize is used internally.

Note: This command does not pass the read size in the queue element but in the command buffer. XPP will look for the size in that buffer.

RBPtr points to the reply buffer. Note that this field will be updated by XPP as it proceeds and will always point to that section of the buffer that XPP is currently reading into.

CCBStart is the start of the memory to be used by the .XPP driver for the command control block. The size of this block is equal to a maximum of 150 bytes. To determine the exact requirement refer to The CCB Sizes section later in this chapter.

Command Block Structure: The AFP read command passes several arguments to XPP in the command buffer itself. The byte offsets are relative to the location pointed to by cbPointer.

-->	0	cmdByte	byte	AFP call command byte
<->	4	rwOffset	long word	Offset within fork to read
<->	8	reqCount	long word	Requested count
-->	12	newLineFlag	byte	Newline Flag
-->	13	newLineChar	byte	Newline Character

CmdByte is the AFP call command byte and must contain the AFP read command code.

RWOffset is the byte offset within the fork at which the read is to begin.

ReqCount indicates the size of the read data buffer and is returned as the actual size read.

The rwOffset and reqCount fields are modified by XPP as the read proceeds and will always indicate the current value of these fields.

NewLineFlag is a one-bit flag (the high bit of the byte) indicating whether or not the read is to terminate at a specified character (all other bits are zero).

0 = no Newline Character is specified
1 = a Newline Character is specified

NewLineChar is any character from $00 to $FF (inclusive) that, when encountered in reading the fork, causes the read operation to terminate.

The Pascal structure of the AFPCommand follows:

```
AFPCommandBlock = PACKED RECORD
                    cmdByte:       Byte;
                    startEndFlag:  Byte;     {unused for read}
                    forkRefNum:    INTEGER;  {used by server}
                    rwOffset:      LONGINT;
                    reqCount:      LONGINT;
                    newLineFlag:   Byte;
                    newLineChar:   CHAR;
                  END;
```

Result codes	aspParamErr	Invalid session number
	aspSizeErr	Command block size is bigger than MaxCmdSize

aspSessClosed	Session is closing
aspBufTooSmall	Reply is bigger than response buffer

CCB Sizes

The .XPP driver uses the memory provided at the end of the UserWrite, UserCommand, and GetStatus functions parameter blocks as an internal command control block (CCB). Using the maximum block sizes specified in the call descriptions will provide adequate space for the call to execute successfully. However, this section is provided for developers who wish to minimize the amount of memory taken up by the CCB in the queue element.

Specifically, this memory is used for building data structures to be used in making calls to the ATP driver. This includes parameter blocks and buffer data structures (BDS). The structure of the BDS is detailed in elsewhere in this chapter. The exact size of this memory depends on the size of the response expected, and, in the case of UserWrite, on the size of data to be written.

In the UserCommand and GetStatus cases (along with all AFP calls which map to UserCommand), a BDS must be set up to hold the response information. The number of entries in this BDS is equal to the size of the response buffer divided by the maximum number of data bytes per ATP response packet (578), rounded up. As described in the ASP chapter in Inside AppleTalk, ASP must ask for an extra response in the case where the response buffer is an exact multiple of 578. Of course, no BDS can be larger than eight elements. XPP also needs bytes for the queue element to call ATP with, so the minimum size of a CCB, as a function of the response buffer size (rbSize) is

```
bdsSize = MIN (((rbSize DIV 578) + 1),8) * bdsEntrySz
ccbSize = ioQElSize + 4 + bdsSize
```

With UserWrite (and AFP calls mapping to UserWrite), XPP must create an additional BDS and queue element to use in sending the write data to the server. Therefore the minimum size of a UserWrite CCB, as a function of the response buffer and write data sizes (rbSize and wdSize) is:

```
wrBDSSize = MIN (((wdSize DIV 578) + 1),8) * bdsEntrySz
wrCCBSize = (2 * ioQElSize) + 4 + bdsSize + wrBDSSize
```

Note: BDSEntrySz is equal to 12; ioQelSize is equal to 50.

.XPP Driver Result Codes

Result Code	Comment	Returned by
aspBadVersNum	Server cannot support the offered version number.	ASPOpenSession AFPCall (Login)
aspBufTooSmall	Reply is bigger than response buffer. Buffer will be filled, data may be truncated.	ASPUserWrite ASPUserCommand ASPGetStatus

		AFPCall
aspNoMoreSess	Driver cannot support another session.	ASPOpenSessION AFPCall (Login)
aspNoServers	No servers at that address. The server did not respond to the request.	ASPGetStatus ASPOpenSession AFPCall (Login)
aspParamErr	Parameter error, server returned bad (positive) error code. Invalid Session Reference Number.	ASPOpenSession ASPCloseSess ASPUserWrite ASPUserCommand AFPCall
aspServerBusy	Server cannot open another session.	ASPOpenSession AFPCall (Login)
aspSessClosed	Session already in process of closing.	ASPCloseSession ASPUserWrite ASPUserCommand AFPCall
aspSizeErr	Command block size is bigger than maxParamSize.	ASPUserWrite ASPUserCommand AFPCall
cbNotFound	SCB not found, no outstanding open session to be aborted. Pointer did not point to an open session SCB.	ASPAbortOS
afpParmError	AFP Command Block size is less than or equal to zero. Command byte in the Command block is equal to 0 or $FF (255) or GetSrvrStatus (15).	AFPCall
reqAborted	Open session was aborted by an Abort Open Session.	ASPOpenSession AFPCall (Login)

PROTOCOL HANDLERS AND SOCKET LISTENERS

This section describes how to write your own protocol handlers and socket listeners. If you're only interested in using the default protocol handlers and socket listeners provided by the Pascal interface, you can skip this section. Protocol handlers and socket listeners must be written in assembly language because they'll be called by the .MPP driver with parameters in various registers not directly accessible from Pascal.

The .MPP and .ATP drivers have been designed to maximize overall throughput while minimizing code size. Two principal sources of loss of throughput are unnecessary buffer copying and inefficient mechanisms for dispatching (routing) packets between the various layers of the network protocol architecture. The AppleTalk Manager completely eliminates buffer copying by using simple, efficient dispatching mechanisms at two important points of the data reception path: protocol handlers and socket listeners. To write your own, you should understand the flow of control in this path.

Data Reception in the AppleTalk Manager

When the SCC detects an ALAP frame addressed to the particular node (or a broadcast frame), it interrupts the Macintosh's MC68000. An interrupt handler built into the .MPP driver gets control and begins servicing the interrupt. Meanwhile, the frame's ALAP header bytes are coming into the SCC's data reception buffer; this is a three-byte FIFO buffer. The interrupt handler must remove these bytes from the SCC's buffer to make room for the bytes right behind; for this purpose, MPP has an internal buffer, known as the Read Header Area (RHA), into which it places these three bytes.

The third byte of the frame contains the ALAP protocol type field. If the most significant bit of this field is set (that is, ALAP protocol types 128 to 255), the frame is an ALAP control frame. Since ALAP control frames are only three bytes long (plus two CRC bytes), for such frames the interrupt handler simply confirms that the CRC bytes indicate an error-free frame and then performs the specified action.

If, however, the frame being received is a data frame (that is, ALAP protocol types 1 to 127), intended for a higher layer of the protocol architecture implemented on that Macintosh, this means that additional data bytes are coming right behind. The interrupt handler must immediately pass control to the protocol handler corresponding to the protocol type specified in the third byte of the ALAP frame for continued reception of the frame. To allow for such a dispatching mechanism, the ALAP code in MPP maintains a protocol table. This consists of a list of currently used ALAP protocol types with the memory addresses of their corresponding protocol handlers. To allow MPP to transfer control to a protocol handler you've written, you must make an appropriate entry in the protocol table with a valid ALAP protocol type and the memory address of your code module.

To enter your protocol handler into the protocol table, issue the LAPOpenProtocol call from Pascal or an AttachPH call from assembly language. Thereafter, whenever an ALAP header with your ALAP protocol type is received, MPP will call your protocol handler. When you no longer wish to receive packets of that ALAP protocol type, call LAPCloseProtocol from Pascal or DetachPH from assembly language.

Warning: Remember that ALAP protocol types 1 and 2 are reserved by DDP for the default protocol handler and that types 128 to 255 are used by ALAP for its control frames.

A protocol handler is a piece of assembly-language code that controls the reception of AppleTalk packets of a given ALAP protocol type. More specifically, a protocol handler must carry out the reception of the rest of the frame following the ALAP header. The nature of a particular protocol handler depends on the characteristics of the protocol for which it was written. In the simplest case, the protocol handler simply reads the entire packet into an internal buffer. A more sophisticated protocol handler might read in the header of its protocol, and on the basis of information contained in it, decide where to put the rest of the packet's data. In certain cases, the protocol handler might, after examining the header corresponding to its own protocol, in turn transfer control to a similar piece of code at the next-higher level of the protocol architecture (for example, in the case of DDP, its protocol handler must call the socket listener of the datagram's destination socket).

In this way, protocol handlers are used to allow "on the fly" decisions regarding the intended recipient of the packets's data, and thus avoid buffer copying. By using protocol handlers and their counterparts in higher layers (for instance, socket listeners), data sent over the AppleTalk network is read directly from the network into the destination's buffer.

Writing Protocol Handlers

When the .MPP driver calls your protocol handler, it has already read the first five bytes of the packet into the RHA. These are the three-byte ALAP header and the next two bytes of the packet. The two bytes following the header must contain the length in bytes of the data in the packet, including these two bytes themselves, but excluding the ALAP header.

Note: Since ALAP packets can have at most 600 data bytes, only the lower ten bits of this length value are significant.

After determining how many bytes to read and where to put them, the protocol handler must call one or both of two functions that perform all the low-level manipulation of the SCC required to read bytes from the network. ReadPacket can be called repeatedly to read in the packet piecemeal or ReadRest can be called to read the rest of the packet. Any number of ReadPacket calls can be used, as long as a ReadRest call is made to read the final piece of the packet. This is necessary because ReadRest restores state information and verifies that the hardware-generated CRC is correct. An error will be returned if the protocol handler attempts to use ReadPacket to read more bytes than remain in the packet.

When MPP passes control to your protocol handler, it passes various parameters and pointers in the processor's registers:

Register(s)	Contents
A0-A1	SCC addresses used by MPP
A2	Pointer to MPP's local variables (discussed below)
A3	Pointer to next free byte in RHA
A4	Pointer to ReadPacket and ReadRest jump table
D1 (word)	Number of bytes left to read in packet

These registers, with the exception of A3, must be preserved until ReadRest is called. A3 is used as an input parameter to ReadPacket and ReadRest, so its contents may be changed. D0, D2, and D3 are free for your use. In addition, register A5 has been saved by MPP and may be used by the protocol handler until ReadRest is called. When control returns to the protocol handler from ReadRest, MPP no longer needs the data in these registers. At that point, standard interrupt routine conventions apply and the protocol handler can freely use A0-A3 and D0-D3 (they're restored by the interrupt handler).

D1 contains the number of bytes left to be read in the packet as derived from the packet's length field. A transmission error could corrupt the length field or some bytes in the packet might be lost, but this won't be discovered until the end of the packet is reached and the CRC checked.

When the protocol handler is first called, the first five bytes of the packet (ALAP destination node ID, source node ID, ALAP protocol type, and length) can be read from the RHA. Since A3 is pointing to the next free position in the RHA, these bytes can be read using negative offsets from A3. For instance, the ALAP source node ID is at –4(A3), the packet's data length (given in D1) is also pointed to by –2(A3), and so on. Alternatively, they can be accessed as positive offsets from the top of the RHA. The effective address of the top of the RHA is toRHA(A2), so the following code could be used to obtain the ALAP type field:

```
LEA       toRHA(A2),A5      ;A5 points to top of RHA
MOVE.B    lapType(A5),D2    ;load D2 with type field
```

These methods are valid only as long as SCC interrupts remain locked out (which they are when the protocol handler is first called). If the protocol handler lowers the interrupt level, another packet could arrive over the network and invalidate the contents of the RHA.

•••Click on the X-Ref button, and refer to Technical Note #201.•••

You can call ReadPacket by jumping through the jump table in the following way:

```
JSR (A4)

On entry    D3:    number of bytes to be read (word)
            A3:    pointer to a buffer to hold the bytes
On exit     D0:    modified
            D1:    number of bytes left to read in packet (word)
            D2:    preserved
            D3:    =  0 if requested number of bytes were read
                   <> 0 if error
```

```
              A0-A2: preserved
              A3:    pointer to one byte past the last byte read
```

ReadPacket reads the number of bytes specified in D3 into the buffer pointed to by A3. The number of bytes remaining to be read in the packet is returned in D1. A3 points to the byte following the last byte read.

You can call ReadRest by jumping through the jump table in the following way:

```
  JSR 2(A4)

  On entry    A3:    pointer to a buffer to hold the bytes
              D3:    size of the buffer (word)
  On exit     D0-D1: modified
              D2:    preserved
              D3:    = 0 if packet was exactly the size of the buffer
                     < 0 if packet was (–D3) bytes too large to fit in
                         buffer and was truncated
                     > 0 if D3 bytes weren't read (packet is smaller
                         than buffer)
              A0-A2: preserved
              A3:    pointer to one byte past the last byte read
```

ReadRest reads the remaining bytes of the packet into the buffer whose size is given in D3 and whose location is pointed to by A3. The result of the operation is returned in D3.

ReadRest can be called with D3 set to a buffer size greater than the packet size; ReadPacket cannot (it will return an error).

Warning: Remember to always call ReadRest to read the last part of a packet; otherwise the system will eventually crash.

If at any point before it has read the last byte of a packet, the protocol handler wants to discard the remaining data, it should terminate by calling ReadRest as follows:

```
  MOVEQ    #0,D3    ;byte count of 0
  JSR      2(A4)    ;call ReadRest
  RTS
```

Or, equivalently:

```
  MOVEQ    #0,D3    ;byte count of 0
  JMP      2(A4)    ;JMP to ReadRest, not JSR
```

In all other cases, the protocol handler should end with an RTS, even if errors were detected. If MPP returns an error from a ReadPacket call, the protocol handler must quit via an RTS without calling ReadRest at all (in this case it has already been called by MPP).

The Z (Zero) condition code is set upon return from these routines to indicate the presence of errors (CRC, overrun, and so on). Zero bit set means no error was detected; a nonzero condition code implies an error of some kind.

Up to 24 bytes of temporary storage are available in MPP's RHA. When the protocol handler is called, 19 of these bytes are free for its use. It may read several bytes (at least four are suggested) into this area to empty the SCC's buffer and buy some time for further processing.

MPP's globals include some variables that you may find useful. They're allocated as a block of memory pointed to by the contents of the global variable ABusVars, but a protocol handler can access them by offsets from A2:

Name	Contents
sysLAPAddr	This node's node ID (byte)
toRHA	Top of the Read Header Area (24 bytes)
sysABridge	Node ID of a bridge (byte)
sysNetNum	This node's network number (word)
vSCCEnable	Status Register (SR) value to re-enable SCC interrupts (word)

Warning: Under no circumstances should your protocol handler modify these variables. It can read them to find the node's ID, its network number, and the node ID of a bridge on the AppleTalk internet.

If, after reading the entire packet from the network and using the data in the RHA, the protocol handler needs to do extensive post-processing, it can load the value in vSCCEnable into the SR to enable interrupts. To allow your programs to run transparently on any Macintosh, use the value in vSCCEnable rather than directly manipulating the interrupt level by changing specific bits in the SR.

Additional information, such as the driver's version number or reference number and a pointer (or handle) to the driver itself, may be obtained from MPP's device control entry. This can be found by dereferencing the handle in the unit table's entry corresponding to unit number 9; for more information, see the section "The Structure of a Device Driver" in the Device Manager chapter.

Timing Considerations

Once it's been called by MPP, your protocol handler has complete responsibility for receiving the rest of the packet. The operation of your protocol handler is time-critical. Since it's called just after MPP has emptied the SCC's three-byte buffer, the protocol handler has approximately 95 microseconds (best case) before it must call ReadPacket or ReadRest. Failure to do so will result in an overrun of the SCC's buffer and loss of packet information. If, within that time, the protocol handler can't determine where to put the entire incoming packet, it should call ReadPacket to read at least four bytes into some private buffer (possibly the RHA). Doing this will again empty the SCC's buffer and buy another 95 microseconds. You can do this as often as necessary, as long as the processing time between successive calls to ReadPacket doesn't exceed 95 microseconds.

Writing Socket Listeners

A socket listener is a piece of assembly-language code that receives datagrams delivered by the DDP built-in protocol handler and delivers them to the client owning that socket.

When a datagram (a packet with ALAP protocol type 1 or 2) is received by the ALAP, DDP's built-in protocol handler is called. This handler reads the DDP header into the RHA, examines the destination socket number, and determines whether this socket is open by searching DDP's socket table. This table lists the socket number and corresponding socket listener address for each open socket. If an entry is found matching the destination socket, the protocol handler immediately transfers control to the appropriate socket listener. (To allow DDP to recognize and branch to a socket listener you've written, call DDPOpenSocket from Pascal or OpenSkt from assembly language.)

At this point, the registers are set up as follows:

Register(s)	Contents
A0-A1	SCC addresses used by MPP
A2	Pointer to MPP's local variables (discussed above)
A3	Pointer to next free byte in RHA
A4	Pointer to ReadPacket and ReadRest jump table
D0	This packet's destination socket number (byte)
D1	Number of bytes left to read in packet (word)

The entire ALAP and DDP headers are in the RHA; these are the only bytes of the packet that have been read in from the SCC's buffer. The socket listener can get the destination socket number from D0 to select a buffer into which the packet can be read. The listener then calls ReadPacket and ReadRest as described under "Writing Protocol Handlers" above. The timing considerations discussed in that section apply as well, as do the issues related to accessing the MPP local variables.

The socket listener may examine the ALAP and DDP headers to extract the various fields relevant to its particular client's needs. To do so, it must first examine the ALAP protocol type field (three bytes from the beginning of the RHA) to decide whether a short (ALAP protocol type=1) or long (ALAP protocol type=2) header has been received.

A long DDP header containing a nonzero checksum field implies that the datagram was checksummed at the source. In this case, the listener can recalculate the checksum using the received datagram, and compare it with the checksum value. The following subroutine can be used for this purpose:

```
DoChksum    ;
            ; D1 (word) = number of bytes to checksum
            ; D3 (word) = current checksum
            ; A1 points to the bytes to checksum
            ;
            CLR.W     D0          ;clear high byte
            SUBQ.W    #1,D1       ;decrement count for DBRA
Loop        MOVE.B    (A1)+,D0    ;read a byte into D0
            ADD.W     D0,D3       ;accumulate checksum
            ROL.W     #1,D3       ;rotate left one bit
            DBRA      D1,Loop     ;loop if more bytes
            RTS
```

Note: D0 is modified by DoChksum.

The checksum must be computed for all bytes starting with the DDP header byte following the checksum field up to the last data byte (not including the CRC bytes). The socket listener must start by first computing the checksum for the DDP header fields in the RHA. This is done as follows:

```
                CLR.W     D3          ;set checksum to 0
                MOVEQ     #ddpHSzLong-ddpDstNet,D1
                                      ;length of header part to checksum
                LEA       toRHA+lapHdSz+ddpDstNet(A2),A1
                                      ;point to destination network number
                JSR       DoChksum
                ; D3 = accumulated checksum of DDP header part
```

The socket listener must now continue to set up D1 and A1 for each subsequent portion of the datagram, and call DoChksum for each. It must not alter the value in D3.

The situation of the calculated checksum being equal to 0 requires special attention. For such packets, the source sends a value of –1 to distinguish them from unchecksummed packets. At the end of its checksum computation, the socket listener must examine the value in D3 to see if it's 0. If so, it's converted to –1 and compared with the received checksum to determine whether there was a checksum error:

```
                TST.W     D3          ;is calculated value 0?
                BNE.S     @1          ;no -- go and use it
                SUBQ.W    #1,D3       ;it is 0; make it -1
  @1            CMP.W     toRHA+lapHdSz+ddpChecksum(A2),D3
                BNE       ChksumError
```

SUMMARY OF THE APPLETALK MANAGER

Constants

```
CONST
  lapSize = 20;   {ABusRecord size for ALAP}
  ddpSize = 26;   {ABusRecord size for DDP}
  nbpSize = 26;   {ABusRecord size for NBP}
  atpSize = 56;   {ABusRecord size for ATP}
```

Data Types

```
TYPE
  ABProtoType = (lapProto,ddpProto,nbpProto,atpProto);
  ABRecHandle = ^ABRecPtr;
  ABRecPtr    = ^ABusRecord;
  ABusRecord  =
    RECORD
       abOpcode:        ABCallType;    {type of call}
       abResult:        INTEGER;       {result code}
       abUserReference: LONGINT;       {for your use}
       CASE ABProtoType OF
         lapProto:
          (lapAddress:  LAPAdrBlock; {destination or source node ID}
           lapReqCount: INTEGER;     {length of frame data or buffer size in }
                                     { bytes}
           lapActCount  INTEGER;     {number of frame data bytes actually }
                                     { received}
           lapDataPtr:  Ptr);        {pointer to frame data or pointer to }
                                     { buffer}
         ddpProto:
          (ddpType:     Byte;        {DDP protocol type}
           ddpSocket:   Byte;        {source or listening socket number}
           ddpAddress:  AddrBlock;   {destination or source socket address}
           ddpReqCount: INTEGER;     {length of datagram data or buffer size }
                                     { in bytes}
           ddpActCount: INTEGER;     {number of bytes actually received}
           ddpDataPtr:  Ptr;         {pointer to buffer}
           ddpNodeID:   Byte);       {original destination node ID}
         nbpProto:
          (nbpEntityPtr:     EntityPtr;    {pointer to entity name}
           nbpBufPtr:        Ptr;          {pointer to buffer}
           nbpBufSize:       INTEGER;      {buffer size in bytes}
           nbpDataField:     INTEGER;      {number of addresses or socket }
                                           { number}
           nbpAddress:       AddrBlock;    {socket address}
           nbpRetransmitInfo:RetransType); {retransmission information}
         atpProto:
          (atpSocket:     Byte;       {listening or responding socket number}
```

```
        atpAddress:    AddrBlock;  {destination or source socket address}
        atpReqCount:   INTEGER;    {request size or buffer size}
        atpDataPtr     Ptr;        {pointer to buffer}
        atpRspBDSPtr:  BDSPtr;     {pointer to response BDS}
        atpBitMap:     BitMapType; {transaction bit map}
        atpTransID:    INTEGER;    {transaction ID}
        atpActCount:   INTEGER;    {number of bytes actually received}
        atpUserData:   LONGINT;    {user bytes}
        atpXO:         BOOLEAN;    {exactly-once flag}
        atpEOM:        BOOLEAN;    {end-of-message flag}
        atpTimeOut:    Byte;       {retry timeout interval in seconds}
        atpRetries:    Byte;       {maximum number of retries}
        atpNumBufs:    Byte;       {number of elements in response BDS or }
                                   { number of response packets sent}
        atpNumRsp:     Byte;       {number of response packets received or }
                                   { sequence number}
        atpBDSSize:    Byte;       {number of elements in response BDS}
        atpRspUData:   LONGINT;    {user bytes sent or received in }
                                   { transaction response}
        atpRspBuf:     Ptr;        {pointer to response message buffer}
        atpRspSize:    INTEGER);   {size of response message buffer}
  END;

ABCallType = (tLAPRead,tLAPWrite,tDDPRead,tDDPWrite,tNBPLookup,tNBPConfirm,
              tNBPRegister,tATPSndRequest,tATPGetRequest,tATPSdRsp,tATPAddRsp,
              tATPRequest,tATPResponse);

LAPAdrBlock = PACKED RECORD
                dstNodeID:    Byte;   {destination node ID}
                srcNodeID:    Byte;   {source node ID}
                lapProtType:  ABByte  {ALAP protocol type}
              END;

ABByte = 1..127; {ALAP protocol type}
AddrBlock = PACKED RECORD
              aNet:     INTEGER;  {network number}
              aNode:    Byte;     {node ID}
              aSocket:  Byte      {socket number}
            END;

BDSPtr     = ^BDSType;
BDSType    = ARRAY[0..7] OF BDSElement; {response BDS}
BDSElement = RECORD
               buffSize:   INTEGER;  {buffer size in bytes}
               buffPtr:    Ptr;      {pointer to buffer}
               dataSize:   INTEGER;  {number of bytes actually received}
               userBytes:  LONGINT   {user bytes}
             END;

BitMapType = PACKED ARRAY[0..7] OF BOOLEAN;
EntityPtr  = ^EntityName;
EntityName = RECORD
               objStr:   Str32;  {object}
               typeStr:  Str32;  {type}
               zoneStr:  Str32    {zone}
```

```
            END;

Str32 = STRING[32];
RetransType =
    PACKED RECORD
      retransInterval:  Byte;  {retransmit interval in 8-tick units}
      retransCount:     Byte   {total number of attempts}
    END;

MPPParamBlock = PACKED RECORD
     qLink:          QElemPtr;     {next queue entry}
     qType:          INTEGER;      {queue type}
     ioTrap:         INTEGER;      {routine trap}
     ioCmdAddr:      Ptr;          {routine address}
     ioCompletion:   ProcPtr;      {completion routine}
     ioResult:       OSErr;        {result code}
     ioNamePtr:      StringPtr;    {command result (ATP user bytes) [long]}
     ioVRefNum:      INTEGER;      {volume reference or drive number}
     ioRefNum:       INTEGER;      {driver reference number}
     csCode:         INTEGER;      {call command code AUTOMATICALLY SET}

     CASE MPPParmType OF
     LAPWriteParm:
                  (filler0:INTEGER;
                  wdsPointer:Ptr);    {->Write Data Structure}
     AttachPHParm,DetachPHParm:
                  (protType:Byte;     {ALAP Protocol Type}
                  filler1:Byte;
                  handler:Ptr);       {->protocol handler routine}
     OpenSktParm,CloseSktParm,WriteDDPParm:
                  (socket:Byte;       {socket number}
                  checksumFlag:Byte;  {checksum flag}
                  listener:Ptr);      {->socket listener routine}
     RegisterNameParm,LookupNameParm,ConfirmNameParm,RemoveNameParm:
                  (interval:Byte;     {retry interval}
                  count:Byte;         {retry count}
                  entityPtr:Ptr;      {->names table element or }
                                      { ->entity name}
                  CASE MPPParmType OF
                  RegisterNameParm:
                           (verifyFlag:Byte;     {set if verify needed}
                            filler3:Byte);
                  LookupNameParm:
                                (retBuffPtr:Ptr;       {->return buffer}
                                retBuffSize:INTEGER;   {return buffer size}
                                maxToGet:INTEGER;      {matches to get}
                                numGotten:INTEGER);    {matched gotten}
                  ConfirmNameParm:
                                (confirmAddr:AddrBlock; {->entity}
                                newSocket:Byte;         {socket number}
                                filler4:Byte));

     SetSelfSendParm:
                  (newSelfFlag:Byte;  {self-send toggle flag}
                  oldSelfFlag:Byte);  {previous self-send state}
```

```
      KillNBPParm:
                  (nKillQEl:Ptr);     {ptr to Q element to cancel}
    END;

ATPParamBlock = PACKED RECORD
        qLink:            QElemPtr;    {next queue entry}
        qType:            INTEGER;     {queue type}
        ioTrap:           INTEGER;     {routine trap}
        ioCmdAddr:        Ptr;         {routine address}
        ioCompletion:     ProcPtr;     {completion routine}
        ioResult:         OSErr;       {result code}
        userData:         LONGINT;     {ATP user bytes [long]}
        reqTID:           INTEGER;     {request transaction ID}
        ioRefNum:         INTEGER;     {driver reference number
        csCode:           INTEGER;     {Call command code }
                                       { AUTOMATICALLY SET}
        atpSocket:        Byte;        {currBitMap or socket number}
        atpFlags:         Byte;        {control information}
        addrBlock:        AddrBlock;   {source/dest. socket address}
        reqLength:        INTEGER;     {request/response length}
        reqPointer:       Ptr;         {-> request/response data}
        bdsPointer:       Ptr;         {-> response BDS}
        CASE MPPParmType OF
              SendRequestParm,NSendRequestParm:
                  (numOfBuffs:Byte;   {numOfBuffs}
                  timeOutVal:Byte;    {timeout interval}
                  numOfResps:Byte;    {number responses actually received}
                  retryCount:Byte;    {number of retries}
                  intBuff:INTEGER);   {used internally for NSendRequest}
              SendResponseParm:
                  (filler0:Byte;      {number of responses being sent}
                  bdsSize:Byte;       {number of BDS elements}
                  transID:INTEGER);   {transaction ID}
              GetRequestParm:
                  (bitMap:Byte;       {bit map}
                  filler1:Byte);
              AddResponseParm:
                  (rspNum:Byte;       {sequence number}
                  filler2:Byte);
              KillSendReqParm,KillGetReqParm:
                  (aKillQEl:Ptr);     {ptr to Q element to cancel}
        END;

  XPPParamBlock = PACKED RECORD
    qLink:          QElemPtr;  {next queue entry}
    qType:          INTEGER;   {queue type}
    ioTrap:         INTEGER;   {routine trap}
    ioCmdAddr:      Ptr;       {routine address}
    ioCompletion:   ProcPtr;   {completion routine}
    ioResult:       OSErr;     {result code}
    cmdResult:      LONGINT;   {command result (ATP user bytes) [long]}
    ioVRefNum:      INTEGER;   {volume reference or drive number)
    ioRefNum:       INTEGER;   {driver reference number)
    csCode:         INTEGER;   {Call command code}
    CASE XPPPrmBlkType OF
```

```
      ASPAbortPrm:
        (abortSCBPtr:    Ptr);      {SCB pointer for AbortOS [long]}
      ASPSizeBlk:
        (aspMaxCmdSize:  INTEGER;   {for SPGetParms [word]
        aspQuantumSize:  INTEGER;   {for SPGetParms [word]}
        numSesss:        INTEGER);  {for SPGetParms [word]}
      XPPPrmBlk:
        (sessRefnum:     INTEGER;   {offset to session refnum [word]}
        aspTimeout:      Byte;      {timeout for ATP [byte]}
        aspRetry:        Byte;      {retry count for ATP [byte]}
        CASE XPPSubPrmType OF
          ASPOpenPrm:
            (serverAddr:    AddrBlock;  {server address block [longword]}
            scbPointer:     Ptr;        {SCB pointer [longword]}
            attnRoutine:    Ptr);       {attention routine pointer [long]}
          ASPSubPrm:
            (cbSize:        INTEGER;    {command block size [word]}
            cbPtr:          Ptr;        {command block pointer [long]}
            rbSize:         INTEGER;    {reply buffer size [word]}
            rbPtr:          Ptr;        {reply buffer pointer [long]}
            CASE XPPEndPrmType OF
              AFPLoginPrm:
                (afpAddrBlock:     AddrBlock;    {address block in}
                                                 { AFPlogin [long]}
                afpSCBPtr:         Ptr;          {SCB pointer in }
                                                 { AFPlogin [long]}
                afpAttnRoutine:    Ptr);         {attn routine pointer }
                                                 { in AFPlogin}
              ASPEndPrm:
                (wdSize:           INTEGER;      {write data size [word]}
                wdPtr:             Ptr;          {write data pointer [long]}
                ccbStart:          ARRAY[0..295] OF Byte)));   {CCB memory }
                                                               { for driver}
    {Write max size(CCB) = 296; all other calls = 150}
    END;

AFPCommandBlock = PACKED RECORD
                    cmdByte:       Byte;
                    startEndFlag:  Byte;
                    forkRefNum:    INTEGER;    {used by server}
                    rwOffset:      LONGINT;
                    reqCount:      LONGINT;
                    newLineFlag:   Byte;       {unused by write}
                    newLineChar:   CHAR;       {unused by write}
                  END;

AFPCommandBlock = PACKED RECORD
                    cmdByte:       Byte;
                    startEndFlag:  Byte;     {unused for read}
                    forkRefNum:    INTEGER;  {used by server}
                    rwOffset:      LONGINT;
                    reqCount:      LONGINT;
                    newLineFlag:   Byte;
                    newLineChar:   CHAR;
                  END;
```

Routines

Opening and Closing AppleTalk

```
FUNCTION MPPOpen :  OSErr;
FUNCTION MPPClose : OSErr;
```

AppleTalk Link Access Protocol

```
FUNCTION LAPOpenProtocol  (theLAPType: ABByte; protoPtr: Ptr) : OSErr;
FUNCTION LAPCloseProtocol (theLAPType: ABByte) : OSErr;

FUNCTION LAPWrite (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
  <--    abOpcode                {always tLAPWrite}
  <--    abResult                {result code}
  -->    abUserReference         {for your use}
  -->    lapAddress.dstNodeID    {destination node ID}
  -->    lapAddress.lapProtType  {ALAP protocol type}
  -->    lapReqCount             {length of frame data}
  -->    lapDataPtr              {pointer to frame data}

FUNCTION LAPRead (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
  <--    abOpcode                {always tLAPRead}
  <--    abResult                {result code}
  -->    abUserReference         {for your use}
  <--    lapAddress.dstNodeID    {destination node ID}
  <--    lapAddress.srcNodeID    {source node ID}
  -->    lapAddress.lapProtType  {ALAP protocol type}
  -->    lapReqCount             {buffer size in bytes}
  <--    lapActCount             {number of frame data bytes actually received}
  -->    lapDataPtr              {pointer to buffer}

FUNCTION LAPRdCancel (abRecord: ABRecHandle) : OSErr;
```

Datagram Delivery Protocol

```
FUNCTION DDPOpenSocket  (VAR theSocket: Byte; sktListener: Ptr) : OSErr;
FUNCTION DDPCloseSocket (theSocket: Byte) : OSErr;

FUNCTION DDPWrite (abRecord: ABRecHandle; doChecksum: BOOLEAN;
                   async:  BOOLEAN) : OSErr;
  <--    abOpcode          {always tDDPWrite}
  <--    abResult          {result code}
  -->    abUserReference   {for your use}
  -->    ddpType           {DDP protocol type}
  -->    ddpSocket         {source socket number}
  -->    ddpAddress        {destination socket address}
  -->    ddpReqCount       {length of datagram data}
  -->    ddpDataPtr        {pointer to buffer}

FUNCTION DDPRead (abRecord: ABRecHandle; retCksumErrs: BOOLEAN;
                  async: BOOLEAN) : OSErr;
```

```
  <--    abOpcode         {always tDDPRead}
  <--    abResult         {result code}
  -->    abUserReference  {for your use}
  <--    ddpType          {DDP protocol type}
  -->    ddpSocket        {listening socket number}
  <--    ddpAddress       {source socket address}
  -->    ddpReqCount      {buffer size in bytes}
  <--    ddpActCount      {number of bytes actually received}
  -->    ddpDataPtr       {pointer to buffer}
  <--    ddpNodeID        {original destination node ID}

FUNCTION DDPRdCancel (abRecord: ABRecHandle) : OSErr;

AppleTalk Transaction Protocol

FUNCTION PNSendRequest (thePBptr: ATPBPtr; async: BOOLEAN) : OSErr;
  -->    18    userData     longword  User bytes
  <--    22    reqTID       word      Transaction ID used in request
  -->    26    csCode       word      Always sendRequest
  <->    28    atpSocket    byte      Socket to send request on
                                        or Current bitmap
  <->    29    atpFlags     byte      Control information
  -->    30    addrBlock    longword  Destination socket address
  -->    34    reqLength    word      Dequest size in bytes
  -->    36    reqPointer   pointer   Pointer to request data
  -->    40    bdsPointer   pointer   Pointer to response BDS
  -->    44    numOfBuffs   byte      Number of responses expected
  -->    45    timeOutVal   byte      Timeout interval
  <--    46    numOf Resps  byte      Number of responses received
  <->    47    retryCount   byte      Number of retries
  <--    48    intBuff      word      Used internally

FUNCTION PKillSendReq (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;
  -->    26    csCode    word     Always PKillSendReq
  -->    44    aKillQEl  pointer  Pointer to queue element

FUNCTION PKillGetReq (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;
  -->    26    csCode    word     Always PKillGetReq
  -->    44    aKillQEl  pointer  Pointer to queue element

FUNCTION ATPLoad :       OSErr;
FUNCTION ATPUnload :     OSErr;
FUNCTION ATPOpenSocket   (addrRcvd: AddrBlock; VAR atpSocket: Byte) : OSErr;
FUNCTION ATPCloseSocket  (atpSocket: Byte) : OSErr;

FUNCTION ATPSndRequest (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
  <--    abOpcode         {always tATPSndRequest}
  <--    abResult         {result code}
  -->    abUserReference  {for your use}
  -->    atpAddress       {destination socket address}
  -->    atpReqCount      {request size in bytes}
  -->    atpDataPtr       {pointer to buffer}
  -->    atpRspBDSPtr     {pointer to response BDS}
  -->    atpUserData      {user bytes}
  -->    atpXO            {exactly-once flag}
```

```
  <--    atpEOM           {end-of-message flag}
  -->    atpTimeOut       {retry timeout interval in seconds}
  -->    atpRetries       {maximum number of retries}
  -->    atpNumBufs       {number of elements in response BDS}
  <--    atpNumRsp        {number of response packets actually received}

FUNCTION ATPRequest (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
  <--    abOpcode         {always tATPRequest}
  <--    abResult         {result code}
  -->    abUserReference  {for your use}
  -->    atpAddress       {destination socket address}
  -->    atpReqCount      {request size in bytes}
  -->    atpDataPtr       {pointer to buffer}
  <--    atpActCount      {number of bytes actually received}
  -->    atpUserData      {user bytes}
  -->    atpXO            {exactly-once flag}
  <--    atpEOM           {end-of-message flag}
  -->    atpTimeOut       {retry timeout interval in seconds}
  -->    atpRetries       {maximum number of retries}
  <--    atpRspUData      {user bytes received in transaction response}
  -->    atpRspBuf        {pointer to response message buffer}
  -->    atpRspSize       {size of response message buffer}

FUNCTION ATPReqCancel (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;

FUNCTION ATPGetRequest (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
  <--    abOpcode         {always tATPGetRequest}
  <--    abResult         {result code}
  -->    abUserReference  {for your use}
  -->    atpSocket        {listening socket number}
  <--    atpAddress       {source socket address}
  -->    atpReqCount      {buffer size in bytes}
  -->    atpDataPtr       {pointer to buffer}
  <--    atpBitMap        {transaction bit map}
  <--    atpTransID       {transaction ID}
  <--    atpActCount      {number of bytes actually received}
  <--    atpUserData      {user bytes}
  <--    atpXO            {exactly-once flag}

FUNCTION ATPSndRsp (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
  <--    abOpcode         {always tATPSdRsp}
  <--    abResult         {result code}
  -->    abUserReference  {for your use}
  -->    atpSocket        {responding socket number}
  -->    atpAddress       {destination socket address}
  -->    atpRspBDSPtr     {pointer to response BDS}
  -->    atpTransID       {transaction ID}
  -->    atpEOM           {end-of-message flag}
  -->    atpNumBufs       {number of response packets being sent}
  -->    atpBDSSize       {number of elements in response BDS}

FUNCTION ATPAddRsp (abRecord: ABRecHandle) : OSErr;
  <--    abOpcode         {always tATPAddRsp}
  <--    abResult         {result code}
  -->    abUserReference  {for your use}
```

```
  -->    atpSocket        {responding socket number}
  -->    atpAddress       {destination socket address}
  -->    atpReqCount      {buffer size in bytes}
  -->    atpDataPtr       {pointer to buffer}
  -->    atpTransID       {transaction ID}
  -->    atpUserData      {user bytes}
  -->    atpEOM           {end-of-message flag}
  -->    atpNumRsp        {sequence number}

FUNCTION ATPResponse (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
  <--    abOpcode         {always tATPResponse}
  <--    abResult         {result code}
  -->    abUserReference  {for your use}
  -->    atpSocket        {responding socket number}
  -->    atpAddress       {destination socket address}
  -->    atpTransID       {transaction ID)
  -->    atpRspUData      {user bytes sent in transaction response}
  -->    atpRspBuf        {pointer to response message buffer}
  -->    atpRspSize       {size of response message buffer}

FUNCTION ATPRspCancel (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
```

Name-Binding Protocol

```
FUNCTION PKillNBP (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;
  -->    26    csCode    word     Always PKillNBP
  -->    28    nKillQEl  pointer  Pointer to queue element

FUNCTION NBPRegister (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
  <--    abOpcode            {always tNBPRegister}
  <--    abResult            {result code}
  -->    abUserReference     {for your use}
  -->    nbpEntityPtr        {pointer to entity name}
  -->    nbpBufPtr           {pointer to buffer}
  -->    nbpBufSize          {buffer size in bytes}
  -->    nbpAddress.aSocket  {socket address}
  -->    nbpRetransmitInfo   {retransmission information}

FUNCTION NBPLookup (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
  <--    abOpcode           {always tNBPLookup}
  <--    abResult           {result code}
  -->    abUserReference    {for your use}
  -->    nbpEntityPtr       {pointer to entity name}
  -->    nbpBufPtr          {pointer to buffer}
  -->    nbpBufSize         {buffer size in bytes}
  <->    nbpDataField       {number of addresses received}
  -->    nbpRetransmitInfo  {retransmission information}

FUNCTION NBPExtract (theBuffer: Ptr; numInBuf: INTEGER; whichOne:  INTEGER;
                     VAR abEntity: EntityName; VAR address:  AddrBlock) : OSErr;

FUNCTION NBPConfirm (abRecord: ABRecHandle; async: BOOLEAN) : OSErr;
  <--    abOpcode           {always tNBPConfirm}
  <--    abResult           {result code}
  -->    abUserReference    {for your use}
```

```
  -->    nbpEntityPtr       {pointer to entity name}
  <--    nbpDataField       {socket number}
  -->    nbpAddress         {socket address}
  -->    nbpRetransmitInfo  {retransmission information}

FUNCTION NBPRemove  (abEntity: EntityPtr) : OSErr;
FUNCTION NBPLoad :   OSErr;
FUNCTION NBPUnload : OSErr;AppleTalk Session Protocol

FUNCTION ASPOpenSession (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;
  -->    26    csCode       word       Always ASPOpenSession
  <--    28    sessRefnum   word       Session reference number
  -->    30    aspTimeout   byte       Retry interval in seconds
  -->    31    aspRetry     byte       Number of retries
  -->    32    serverAddr   long word  Server socket address
  -->    36    scbPointer   pointer    Pointer to session control block
  -->    40    attnRoutine  pointer    Pointer to attention routine

FUNCTION ASPCloseSession (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;
  -->    26    csCode      word  Always ASPCloseSess
  -->    28    sessRefnum  word  Session reference number

FUNCTION ASPAbortOS (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;
  -->    26    csCode           word     Always ASPAbortOS
  -->    28    abortSCBPointer  pointer  Pointer to session control block

FUNCTION ASPGetParms (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;
  -->    26    csCode          word  Always ASPGetParms
  <--    28    aspMaxCmdSize   word  Maximum size of command block
  <--    30    aspQuantumSize  word  Maximum data size
  <--    32    numSesss        word  Number of sessions

FUNCTION ASPCloseAll (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;
  -->    26    csCode  word  Always ASPCloseAll

FUNCTION ASPUserWrite (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;
  <--    18    cmdResult   long word  ASP command result
  -->    26    csCode      word       Always ASPUserWrite
  -->    28    sessRefnum  word       Session reference number
  -->    30    aspTimeout  byte       Retry interval in seconds
  -->    32    cbSize      word       Command block size
  -->    34    cbPtr       pointer    Command block pointer
  <->    38    rbSize      word       Reply buffer size and reply size
  -->    40    rbPointer   pointer    Reply buffer pointer
  <->    44    wdSize      word       Write data size
  -->    46    wdPtr       pointer    Write data pointer
  <--    50    ccbStart    record     Start of memory for CCB

FUNCTION ASPUserCommand (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;
  <--    18    cmdResult   long word  ASP command result
  <--    26    csCode      word       Always ASPUserCommand
  <--    28    sessRefnum  word       Session number
  <--    30    aspTimeout  byte       Retry interval in seconds
  <--    32    cbSize      word       Command block size
  <--    34    cbPtr       pointer    Command block pointer
```

```
  <->    38    rbSize      word       Reply buffer and reply size
  <--    40    rbPtr       pointer    Reply buffer pointer
  <--    50    ccbStart    record     Start of memory for CCB

FUNCTION ASPGetStatus (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;
  <--    26    csCode      word       Always ASPGetStatus
  <--    30    aspTimeout  byte       Retry interval in seconds
  <--    31    aspRetry    byte       Number of retries
  <--    32    serverAddr  long word  Server socket address
  <->    38    rbSize      word       Reply buffer and reply size
  <--    40    rbPtr       pointer    Reply buffer pointer
  <--    50    ccbStart    record     Start of memory for CCB
```

General Command Format

```
FUNCTION AFPCommand (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;
```

Parameter block

```
  <--    18    cmdResult   long word  AFP command result
  <--    26    csCode      word       Always AFPCall
  <--    28    sessRefnum  word       Session reference number
  <--    30    aspTimeout  byte       Retry interval in seconds
  <--    32    cbSize      word       Command buffer size
  <--    34    cbPtr       pointer    Command buffer
  <->    38    rbSize      word       Reply buffer size and reply size
  <--    40    rbPtr       pointer    Reply buffer pointer
  <->    44    wdSize      word       Write data size
  <--    46    wdPtr       pointer    Write data pointer
  <--    50    ccbStart    record     Start of memory for CCB
```

Login Command Format

```
FUNCTION AFPCommand (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;
  <--    18    cmdResult       long word  AFP command result
  <--    26    csCode          word       Always AFPCall
  <--    28    sessRefnum      word       Session reference number
  <--    30    aspTimeout      byte       Retry interval in seconds
  <--    31    aspRetry        byte       Number of retries
  <--    32    cbSize          word       Command buffer size
  <--    34    cbPtr           pointer    Command buffer
  <->    38    rbSize          word       Reply buffer size and reply size
  <--    40    rbPtr           pointer    Reply buffer pointer
  <--    44    afpAddrBlock    long word  Server address block
  <->    48    afpSCBPtr       pointer    SCB pointer
  <->    52    afpAttnRoutine  pointer    Attention routine pointer
  <--    50    ccbStart        record     Start of command control block
```

AFPWrite Command Format

```
FUNCTION AFPCommand (xParamBlock: XPPParmBlkPtr; async: BOOLEAN) : OSErr;
  <--    18    cmdResult   long word  AFP command result
  <--    26    csCode      word       Always AFPCall
  <--    28    sessRefnum  word       Session number
  <--    30    aspTimeout  byte       Retry interval in seconds
  <--    32    cbSize      word       Command buffer size
```

```
  <--    34    cbPtr       pointer    Command buffer
  <->    38    rbSize      word       Reply buffer size and reply size
  <--    40    rbPtr       pointer    Reply buffer pointer
  <--    44    wdSize      word       (used internally)
  <->    46    wdPtr       pointer    Write data pointer  (updated)
  <--    50    ccbStart    record     Start of memory for CCB
```

Command Block Structure

```
  <--    0    cmdByte       byte       AFP call command byte
  <--    1    startEndFlag  byte       Start/end Flag
  <->    4    rwOffset      long word  Offset within fork to write
  <->    8    reqCount      long word  Requested count
```

AFPRead Command Format

```
FUNCTION AFPCommand (xParamBlock: XPPParmBlkPtr; async: BOOLEAN): OSErr;
  <--    18    cmdResult   long word  ASP command result
  <--    26    csCode      word       Always AFPCall
  <--    28    sessRefnum  word       Session number
  <--    30    aspTimeout  byte       Retry interval in seconds
  <--    32    cbSize      word       Command buffer size
  <--    34    cbPtr       pointer    Command buffer
  <--    38    rbSize      word       Used internally
  <->    40    rbPtr       pointer    Reply buffer pointer (updated)
  <--    50    ccbStart    record     Start of memory for CCB
```

Command Block Structure

```
  <--    0    cmdByte      byte       AFP call command byte
  <->    4    rwOffset     long word  Offset within fork to read
  <->    8    reqCount     long word  Requested count
  <--    12   newLineFlag  byte       Newline Flag
  <--    13   newLineChar  byte       Newline Character
```

Miscellaneous Routines

```
FUNCTION GetNodeAddress (VAR myNode,myNet: INTEGER) : OSErr;
FUNCTION IsMPPOpen :     BOOLEAN;
FUNCTION IsATPOpen :     BOOLEAN;

FUNCTION PSetSelfSend (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;
  -->    26    csCode       word  Always PSetSelfSend
  -->    28    newSelfFlag  byte  New SelfSend flag
  <--    29    oldSelfFlag  byte  Old SelfSend flag
```

Result Codes

Name	Value	Meaning
atpBadRsp	–3107	Bad response from ATPRequest
atpLenErr	–3106	ATP response message too large
badATPSkt	–1099	ATP bad responding socket
badBuffNum	–1100	ATP bad sequence number

```
buf2SmallErr    –3101    ALAP frame too large for buffer DDP datagram
                         too large for buffer
cbNotFound      –1102    ATP control block not found
cksumErr        –3103    DDP bad checksum
ddpLenErr         –92    DDP datagram or ALAP data length too big
ddpSktErr         –91    DDP socket error: socket already active; not a
                         well-known socket; socket table full; all dynamic
                         socket numbers in use
excessCollsns     –95    ALAP no CTS received after 32 RTS's, or line
                         sensed in use 32 times (not necessarily caused
                         by collisions)
extractErr      –3104    NBP can't find tuple in buffer
lapProtErr        –94    ALAP error attaching/detaching ALAP protocol type:
                         attach error when ALAP protocol type is negative,
                         not in range, already in table, or when table is full;
                         detach error when ALAP protocol type isn't in table
nbpBuffOvr      –1024    NBP buffer overflow
nbpConfDiff     –1026    NBP name confirmed for different socket
nbpDuplicate    –1027    NBP duplicate name already exists
nbpNISErr       –1029    NBP names information socket error
nbpNoConfirm    –1025    NBP name not confirmed
nbpNotFound     –1028    NBP name not found
noBridgeErr       –93    No bridge found
noDataArea      –1104    Too many outstanding ATP calls
noErr              0     No error
noMPPError      –3102    MPP driver not installed
noRelErr        –1101    ATP no release received
noSendResp      –1103    ATPAddRsp issued before ATPSndRsp
portInUse         –97    Driver Open error, port already in use
portNotCf         –98    Driver Open error, port not configured
                         for this connection
readQErr        –3105    Socket or protocol type invalid or not found in table
recNotFnd       –3108    ABRecord not found
reqAborted      –1105    Request aborted
reqFailed       –1096    ATPSndRequest failed: retry count exceeded
sktClosedErr    –3109    Asynchronous call aborted because socket was
                         closed before call was completed
tooManyReqs     –1097    ATP too many concurrent requests
tooManySkts     –1098    ATP too many responding sockets
```

Assembly-Language Information

Constants

```
; Serial port use types

useFree       .EQU    0     ;unconfigured
useATalk      .EQU    1     ;configured for AppleTalk
useASync      .EQU    2     ;configured for the Serial Driver

; Bit in PortBUse for .ATP driver status

atpLoadedBit  .EQU    4     ;set if .ATP driver is opened
```

```
; Unit numbers for AppleTalk drivers

mppUnitNum    .EQU    9     ;.MPP driver
atpUnitNum    .EQU    10    ;.ATP driver

; csCode values for Control calls (MPP)

writeLAP      .EQU    243
detachPH      .EQU    244
attachPH      .EQU    245
writeDDP      .EQU    246
closeSkt      .EQU    247
openSkt       .EQU    248
loadNBP       .EQU    249
confirmName   .EQU    250
lookupName    .EQU    251
removeName    .EQU    252
registerName  .EQU    253
killNBP       .EQU    254
unloadNBP     .EQU    255

; csCode values for Control calls (ATP)

relRspCB      .EQU    249
closeATPSkt   .EQU    250
addResponse   .EQU    251
sendResponse  .EQU    252
getRequest    .EQU    253
openATPSkt    .EQU    254
sendRequest   .EQU    255
relTCB        .EQU    256

; ALAP header

lapDstAdr     .EQU    0     ;destination node ID
lapSrcAdr     .EQU    1     ;source node ID
lapType       .EQU    2     ;ALAP protocol type

; ALAP header size

lapHdSz       .EQU    3

; ALAP protocol type values

shortDDP      .EQU    1     ;short DDP header
longDDP       .EQU    2     ;long DDP header

; Long DDP header

ddpHopCnt     .EQU    0     ;count of bridges passed (4 bits)
ddpLength     .EQU    0     ;datagram length (10 bits)
ddpChecksum   .EQU    2     ;checksum
ddpDstNet     .EQU    4     ;destination network number
ddpSrcNet     .EQU    6     ;source network number
```

```
ddpDstNode    .EQU    8     ;destination node ID
ddpSrcNode    .EQU    9     ;source node ID
ddpDstSkt     .EQU    10    ;destination socket number
ddpSrcSkt     .EQU    11    ;source socket number
ddpType       .EQU    12    ;DDP protocol type

; DDP long header size

ddpHSzLong    .EQU    ddpType+1

; Short DDP header

ddpLength     .EQU    0               ;datagram length
sDDPDstSkt    .EQU    ddpChecksum     ;destination socket number
sDDPSrcSkt    .EQU    sDDPDstSkt+1    ;source socket number
sDDPType      .EQU    sDDPSrcSkt+1    ;DDP protocol type

; DDP short header size

ddpHSzShort   .EQU    sDDPType+1

; Mask for datagram length

ddpLenMask    .EQU    $03FF

; Maximum size of DDP data

ddpMaxData    .EQU    586

; ATP header

atpControl    .EQU    0     ;control information
atpBitMap     .EQU    1     ;bit map
atpRespNo     .EQU    1     ;sequence number
atpTransID    .EQU    2     ;transaction ID
atpUserData   .EQU    4     ;user bytes

; ATP header size

atpHdSz       .EQU    8

; DDP protocol type for ATP packets

atp           .EQU    3

; ATP function code

atpReqCode    .EQU    $40    ;TReq packet
atpRspCode    .EQU    $80    ;TResp packet
atpRelCode    .EQU    $C0    ;TRel packet

; ATPFlags control information bits

sendChk       .EQU    0     ;send-checksum bit
tidValid      .EQU    1     ;transaction ID validity bit
```

```
atpSTSBit     .EQU    3     ;send-transmission-status bit
atpEOMBit     .EQU    4     ;end-of-message bit
atpXOBit      .EQU    5     ;exactly-once bit

; Maximum number of ATP request packets

atpMaxNum     .EQU    8

; ATP buffer data structure

bdsBuffSz     .EQU    0     ;size of data to send or buffer size
bdsBuffAddr   .EQU    2     ;pointer to data or buffer
bdsDataSz     .EQU    6     ;number of bytes actually received
bdsUserData   .EQU    8     ;user bytes

; BDS element size

bdsEntrySz    .EQU    12

; NBP packet

nbpControl    .EQU    0     ;packet type
nbpTCount     .EQU    0     ;tuple count
nbpID         .EQU    1     ;packet identifier
nbpTuple      .EQU    2     ;start of first tuple

; DDP protocol type for NBP packets

nbp           .EQU    2

; NBP packet types

brRq          .EQU    1     ;broadcast request
lkUp          .EQU    2     ;lookup request
lkUpReply     .EQU    3     ;lookup reply

; NBP tuple

tupleNet      .EQU    0     ;network number
tupleNode     .EQU    2     ;node ID
tupleSkt      .EQU    3     ;socket number
tupleEnum     .EQU    4     ;used internally
tupleName     .EQU    5     ;entity name

; Maximum number of tuples in NBP packet

tupleMax      .EQU    15

; NBP meta-characters

equals        .EQU    '='    ;"wild-card" meta-character
star          .EQU    '*'    ;"this zone" meta-character

; NBP names table entry
```

```
ntLink        .EQU    0     ;pointer to next entry
ntTuple       .EQU    4     ;tuple
ntSocket      .EQU    7     ;socket number
ntEntity      .EQU    9     ;entity name

; NBP names information socket number

nis           .EQU    2

Offsets in User Bytes

aspCmdCode      EQU     0      ;offset to command field
aspWSSNum       EQU     1      ;WSS number in OpenSessions
aspVersNum      EQU     2      ;ASP version number in OpenSessions
aspSSSNum       EQU     0      ;SSS number in OpenSessReplies
aspSessID       EQU     1      ;session ID (requests &OpenSessReply)
aspOpenErr      EQU     2      ;OpenSessReply error code

aspSeqNum       EQU     2      ;sequence number in requests
aspAttnCode     EQU     2      ;attention bytes in attentions

Offsets in ATP data part

aspWrBSize      EQU     0             ;offset to write buffer size (WriteData)
aspWrHdrSz      EQU     ASPWrBSize+2  ;size of data part

ASP command codes

aspCloseSess    EQU     1       ;close session
aspCommand      EQU     2       ;user-command
aspGetStat      EQU     3       ;get status
aspOpenSess     EQU     4       ;open session
aspTickle       EQU     5       ;tickle
aspWrite        EQU     6       ;write
aspDataWrite    EQU     7       ;writedata (from server)
aspAttention    EQU     8       ;attention (from server)

ASP miscellaneous

aspVersion      EQU     $0100                 ;ASP version number
MaxCmdSize      EQU     ATPMaxData            ;maximum command block size
QuantumSize     EQU     ATPMaxData*ATPMaxNum  ;maximum reply size
XPPLoadedBit    EQU     ATPLoadedBit+1        ;XPP bit in PortBUse
XPPUnitNum      EQU     40                    ;unit number for XPP (old ROMs)

ASP errors codes

aspBadVersNum   EQU  -1066  ;server cannot support this ASP version
aspBufTooSmall  EQU  -1067  ;buffer too small
aspNoMoreSess   EQU  -1068  ;no more sessions on server
aspNoServers    EQU  -1069  ;no servers at that address
aspParamErr     EQU  -1070  ;parameter error
aspServerBusy   EQU  -1071  ;server cannot open another session
aspSessClosed   EQU  -1072  ;session closed
aspSizeErr      EQU  -1073  ;command block too big
```

```
aspTooMany      EQU  -1074  ;too many clients
aspNoAck        EQU  -1075  ;no ack on attention Request

Control codes

openSess       EQU    255    ;open session
closeSess      EQU    254    ;close session
userCommand    EQU    253    ;user command
userWrite      EQU    252    ;user write
getStatus      EQU    251    ;get status
afpCall        EQU    250    ;AFP command (buffer has command code)
getParms       EQU    249    ;get parameters
abortOS        EQU    248    ;abort open session request
closeAll       EQU    247    ;close all open sessions

ASP queue element standard structure:  arguments passed in the CSParam area

sessRefnum      EQU     $1C   ;offset to session refnum [word]
aspTimeout      EQU     $1E   ;timeout for ATP [byte]
aspRetry        EQU     $1F   ;retry count for ATP [byte]
serverAddr      EQU     $20   ;server address block [longword]
scbPointer      EQU     $24   ;SCB pointer [longword]
attnRoutine     EQU     $28   ;attention routine pointer [long]

cbSize          EQU     $20   ;command block size [word]
cbPtr           EQU     $22   ;command block pointer [long]
rbSize          EQU     $26   ;reply buffer size [word]
rbPtr           EQU     $28   ;reply buffer pointer [long]
wdSize          EQU     $2C   ;write data size [word]
wdPtr           EQU     $2E   ;write data pointer [long]
ccbStart        EQU     $32   ;start of memory for CCB

aspMaxCmdSize   EQU     $1C   ;for SPGetParms [word]
aspQuantumSize  EQU     $1E   ;for SPGetParms [word]
abortSCBPtr     EQU     $1F   ;SCB pointer for AbortOS [long]

cmdResult       EQU     $12   ;command result (ATP user
                              ; bytes)[long]

afpAddrBlock    EQU     $2C   ;address block in AFP login[long]
afpSCBPtr       EQU     $30   ;SCB pointer in AFP login [long]
afpAttnRoutine  EQU     $34   ;attn routine pointer in AFP login

scbMemSize      EQU     $C0   ;size of memory for SCB

AFPCall command codes

afpLogin           EQU   18;
afpContLogin       EQU   19;
afpLogout          EQU   20;
afpRead            EQU   27;
afpWrite           EQU   33;

Offsets for certain parameters in Read/Write calls
```

```
startEndFlag   EQU    $1 ;write only; offset relative to start or end
rwOffset       EQU    $4 ;offset at which to start read or write
reqCount       EQU    $8 ;count of bytes to read or write
newLineFlag    EQU    $C ;read only; newline character flag
newLineChar    EQU    $D ;read only; newline character
lastWritten    EQU    $0 ;write only; last written  (returned)

Miscellaneous

afpUseWrite  EQU  $C0  ;first call in range that maps to an
                       ; ASPWrite

Routines

Preferred Interface Routines

AttachPH     Function PAttachPH    (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;
DetachPH     Function PDetachPH    (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;
WriteLAP     Function PWriteLAP    (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;
OpenSkt      Function POpenSkt     (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;
CloseSkt     Function PCloseSkt    (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;
WriteDDP     Function PWriteDDP    (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;
RegisterName Function PRegisterName(thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;
LookupName   Function PLookupName  (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;
ConfirmName  Function PConfirmName (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;
RemoveName   Function PRemoveName  (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;
OpenATPSkt   Function POpenATPSkt  (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;
CloseATPSkt  Function PCloseATPSkt (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;
SendRequest  Function PSendRequest (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;
GetRequest   Function PGetRequest  (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;
SendResponse Function PSendResponse(thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;
AddResponse  Function PAddResponse (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;
ReLTCB       Function PRelTCB      (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;
RelRspCB     Function PRelRspCB    (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;
SetSelfSend  Function PSetSelfSend (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;
NSendRequest Function PNSendRequest(thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;
KillSendReq  Function PKillSendReq (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;
KillGetReq   Function PKillGetReq  (thePBptr: ATPPBPtr; async: BOOLEAN) : OSErr;
KillNBP      Function PKillNBP     (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;

PROCEDURE BuildLAPwds (wdsPtr,dataPtr: Ptr;
                       destHost,protoType,frameLen: INTEGER);
PROCEDURE BuildDDPwds  (wdsPtr,headerPtr,dataPtr: Ptr; destAddress: AddrBlock;
                        DDPType : INTEGER; dataLen: INTEGER);
PROCEDURE NBPSetEntity (buffer: Ptr; nbpObject,nbpType,nbpZone: Str32);
PROCEDURE NBPSetNTE (ntePtr: Ptr; nbpObject,nbpType,nbpZone: Str32;
                     Socket: INTEGER);
FUNCTION  NBPExtract (theBuffer: Ptr; numInBuf: INTEGER; whichOne: INTEGER;
                      VAR abEntity: EntityName; VAR address: AddrBlock) : OSErr;
FUNCTION  GetBridgeAddress: INTEGER;
FUNCTION  BuildBDS (buffPtr,bdsPtr: Ptr; buffSize: INTEGER) : INTEGER;

Alternate Interface Routines

Link Access Protocol
```

```
WriteLAP function
  -->    26    csCode      word     ;always writeLAP
  -->    30    wdsPointer  pointer  ;write data structure

AttachPH function
  -->    26    csCode    word     ;always attachPH
  -->    28    protType  byte     ;ALAP protocol type
  -->    30    handler   pointer  ;protocol handler

DetachPH function
  -->    26    csCode    word  ;always detachPH
  -->    28    protType  byte  ;ALAP protocol type

Datagram Delivery Protocol

OpenSkt function
  -->    26    csCode    word     ;always openSkt
  <->    28    socket    byte     ;socket number
  -->    30    listener  pointer  ;socket listener

CloseSkt function
  -->    26    csCode  word  ;always closeSkt
  -->    28    socket  byte  ;socket number

WriteDDP function
  -->    26    csCode        word     ;always writeDDP
  -->    28    socket        byte     ;socket number
  -->    29    checksumFlag  byte     ;checksum flag
  -->    30    wdsPointer    pointer  ;write data structure

AppleTalk Transaction Protocol

OpenATPSkt function
  -->    26    csCode     word       ;always openATPSkt
  <->    28    atpSocket  byte       ;socket number
  -->    30    addrBlock  long word  ;socket request specification

CloseATPSkt function
  -->    26    csCode     word  ;always closeATPSkt
  -->    28    atpSocket  byte  ;socket number

SendRequest function
  -->    18    userData    long word  ;user bytes
  <--    22    reqTID      word       ;transaction ID used in request
  -->    26    csCode      word       ;always sendRequest
  <--    28    currBitMap  byte       ;bit map
  <->    29    atpFlags    byte       ;control information
  -->    30    addrBlock   long word  ;destination socket address
  -->    34    reqLength   word       ;request size in bytes
  -->    36    reqPointer  pointer    ;pointer to request data
  -->    40    bdsPointer  pointer    ;pointer to response BDS
  -->    44    numOfBuffs  byte       ;number of responses expected
  -->    45    timeOutVal  byte       ;timeout interval
  <--    46    numOfResps  byte       ;number of responses received
```

```
  <->    47    retryCount  byte       ;number of retries

GetRequest function
  <--    18    userData    long word  ;user bytes
  -->    26    csCode      word       ;always getRequest
  -->    28    atpSocket   byte       ;socket number
  <--    29    atpFlags    byte       ;control information
  <--    30    addrBlock   long word  ;source of request
  <->    34    reqLength   word       ;request buffer size
  -->    36    reqPointer  pointer    ;pointer to request buffer
  <--    44    bitMap      byte       ;bit map
  <--    46    transID     word       ;transaction ID

SendResponse function
  <--    18    userData    long word  ;user bytes from TRel
  -->    26    csCode      word       ;always sendResponse
  -->    28    atpSocket   byte       ;socket number
  -->    29    atpFlags    byte       ;control information
  -->    30    addrBlock   long word  ;response destination
  -->    40    bdsPointer  pointer    ;pointer to response BDS
  -->    44    numOfBuffs  byte       ;number of response packets being sent
  -->    45    bdsSize     byte       ;BDS size in elements
  -->    46    transID     word       ;transaction ID

AddResponse function
  -->    18    userData    long word  ;user bytes
  -->    26    csCode      word       ;always addResponse
  -->    28    atpSocket   byte       ;socket number
  -->    29    atpFlags    byte       ;control information
  -->    30    addrBlock   long word  ;response destination
  -->    34    reqLength   word       ;response size
  -->    36    reqPointer  pointer    ;pointer to response
  -->    44    rspNum      byte       ;sequence number
  -->    46    transID     word       ;transaction ID

RelTCB function
  -->    26    csCode     word       ;always relTCB
  -->    30    addrBlock  long word  ;destination of request
  -->    46    transID    word       ;transaction ID of request

RelRspCB function
  -->    26    csCode     word       ;always relRspCB
  -->    28    atpSocket  byte       ;socket number that request was
                                     ; received on
  -->    30    addrBlock  long word  ;source of request
  -->    46    transID    word       ;transaction ID of request

Name-Binding Protocol

RegisterName function
  -->    26    csCode      word     ;always registerName
  -->    28    interval    byte     ;retry interval
  <->    29    count       byte     ;retry count
  -->    30    ntQElPtr    pointer  ;names table element pointer
  -->    34    verifyFlag  byte     ;set if verify needed
```

```
LookupName function
  -->    26    csCode       word     ;always lookupName
  -->    28    interval     byte     ;retry interval
  <->    29    count        byte     ;retry count
  -->    30    entityPtr    pointer  ;pointer to entity name
  -->    34    retBuffPtr   pointer  ;pointer to buffer
  -->    38    retBuffSize  word     ;buffer size in bytes
  -->    40    maxToGet     word     ;matches to get
  <--    42    numGotten    word     ;matches found

ConfirmName function
  -->    26    csCode       word     ;always confirmName
  -->    28    interval     byte     ;retry interval
  <->    29    count        byte     ;retry count
  -->    30    entityPtr    pointer  ;pointer to entity name
  -->    34    confirmAddr  pointer  ;entity address
  <--    38    newSocket    byte     ;socket number

RemoveName function
  -->    26    csCode     word     ;always removeName
  -->    30    entityPtr  pointer  ;pointer to entity name

LoadNBP function
  -->    26    csCode  word  ;always loadNBP

UnloadNBP function
  -->    26    csCode  word  ;always unloadNB

Variables

SPConfig    Use types for serial ports (byte)
            (bits 0-3:  current configuration of serial port B
             bits 4-6:  current configuration of serial port A)
PortBUse    Current availability of serial port B (byte)
            (bit 7:  1 = not in use, 0 = in use bits
             bits 0-3:  current use of port bits
             bits 4-6:  driver-specific)
ABusVars    Pointer to AppleTalk variables
```

Further Reference:

Toolbox Event Manager
Device Manager
Technical Note #9, Will Your AppleTalk Application Support Internets?
Technical Note #20, Data Servers on AppleTalk
Technical Note #121, Using the High-Level AppleTalk Routines
Technical Note #132, AppleTalk Interface Update
Technical Note #142, Avoid Use of Network Events
Technical Note #195, ASP and AFP Description Discrepancies
Technical Note #199, KillNBP Clarification
Technical Note #201, ReadPacket Clarification
Technical Note #224, Opening AppleTalk
Technical Note #225, Using RegisterName
Technical Note #250, AppleTalk Phase 2 on the Macintosh
"Inside AppleTalk"

END OF DOCUMENT